Increasing Usability of Spin-based C Code Verification Using a Harness Definition Language

Leveraging Model-driven Code Checking to Practitioners

Daniel Ratiu
Siemens Corporate Technology
Otto-Hahn-Ring 6
81739 Munich, Germany
daniel.ratiu@siemens.com

Andreas Ulrich
Siemens Corporate Technology
Otto-Hahn-Ring 6
81739 Munich, Germany
andreas.ulrich@siemens.com

ABSTRACT

Due to its capabilities to integrate well with C code, Spin has been used for C code verification based on environment models defined in Promela that describe the context, in which the software under verification is expected to run. In practice this approach requires an in-depth knowledge of Promela and the underlying technology. Moreover environment models tend to be verbose and exhibit heavily intertwined statements of Promela and C code. Thereby, writing and understanding such hybrid models is difficult and error-prone. Alleviating this problem we develop a specialized language, based on Promela, for expressing environment models used in verification harnesses. Our language harmonizes the use of Promela and C in a homogeneous way that is suitable for practitioners. We show how a small number of language concepts is sufficient to define environments for a wide variety of commonly encountered software components written in C. The approach is integrated in the development platform mbeddr, a technology stack for embedded programming and formal verification developed on top of JetBrains' MPS.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Model Checking; D.2.5 [**Software and its engineering**]: Software testing and debugging

Keywords

Spin, domain-specific languages, model checking, testing

1. INTRODUCTION

Formal verification increases the quality of software by enabling the engineers to discover subtle bugs. Despite dramatic improvements of the verification algorithms and tooling in the last 10 years, software verification tools are regarded as expert tools and developers shy away in front of their (perceived) complexity.

While software verification is conventionally applied on the source code of the *Software Under Verification* (SUV), an alternative approach based on Spin, a highly mature explicit state model-checker, has been proposed in [11]. In this scenario, a verification model is not created from the SUV directly, but from its environment, which the SUV interacts with. Taking advantage of the possibility to embed C code in Promela, Spin generates a verifier that systematically explores the states of the environment model and calls the SUV. A prerequisite of this approach is that calls to the SUV are atomic, i.e. there are no intermediate states in the environment model between the call and the return. For this reason, the approach is best applicable in the context of unit testing, treating the SUV as a black-box. The approach is therefore complementary to white-box testing approaches such as KLEE/LLVM [4] that generate tests from the source code of the SUV with the purpose of maximising code coverage during test execution.

The environment-driven black-box approach to code checking draws its advantage from the fact that the SUV is called in the context of its usage — provided that the environment model faithfully expresses this usage. The engineer has a full range of possibilities at hand: from a trivial model with a single state and no constraints on the SUV input data (the most general model) to a highly elaborated state model with constraint data sets. This way the engineer can control the extend to which the SUV is analyzed at runtime.

Deploying Spin for the exploration of the environment model requires, naturally, the use of Promela as its specification language. C code developers who are not trained in formal verification find it hard to apply Promela with its specific verification features for this purpose, rendering this approach quickly unacceptable in practice. In a long-term pursuit, Siemens invests into technologies to make formal verification techniques usable up to a level that common practitioners can benefit from its potential. The approach outlined in this paper is based on language engineering technologies to hide the complexity of using formal methods directly [15, 14]. The entire approach is implemented in mbeddr [19][1], an integrated development environment (IDE) and open source stack of domain specific languages on top of C to assist the development of safety-critical C code.

[1] www.mbeddr.com

```
1  c_decl { uchar my_array[4], *res; }
2
3  active proctype sort_harness() {
4    byte elem0, elem1, elem2, elem3;
5    select(elem0 : 0 .. 255);
6    ...
7    select(elem3 : 0 .. 255);
8    c_code {
9      my_array[0] = Psort_harness->elem0;
10     ...
11     my_array[3] = Psort_harness->elem3;
12     res = sort(my_array, 4);
13     for(i = 0; i < 3; i++) {
14       if (!(res[i] <= res[i+1])) {
15         uerror("...");
16 } } } }
```

```
1  decls { uchar my_array[4], *res; }
2
3  harness sort {
4
5    nondet_assign(my_array, 0, 255);
6
7
8
9
10
11
12   res = sort(my_array, 4);
13   for(i = 0; i < 3; i++) {
14     assert(res[i] <= res[i+1]);
15   }
16 }
```

Figure 1: Verification harness definition using Promela (left) and a proposed higher-level DSL (right)

To outline our language engineering approach, consider Figure 1 that sketches the approach of environment-driven code checking using a sorting algorithm called `sort()` as the SUV. The environment model, called *verification harness* in the remainder of the text, `sort_harness()` is provided as a Promela model (left-hand side). The model is conceptually simple. It generates valid input data through a non-deterministic assignment of values to the array to be sorted (lines 4-11), calls the `sort()` function (line 12) and checks that the array returned is correctly sorted (lines 13-17). Note the mixed usage of Promela statements and embedded C code statements. The right-hand side illustrates our proposed domain-specific language (DSL) which hides the repetitive parts in the Promela model and offers a simple and direct way to describe the verification harness by blending the distinction between C code and Promela models.

The work presented enhances the model-driven code checking approach [11] with the following aspects: 1) We define a small set of concepts of a harness definition language which are conceptually simple to use, yet powerful enough to describe a wide variety of verification harnesses; 2) We show how these constructs are encoded in Promela to derive executable environment models that are further processed by Spin; 3) We develop an enhanced way to express the witness to ease their understanding, and 4) We describe mbeddr-spin, a tool that implements our approach.

The remainder of the paper is organized as follows. In Section 2 we describe a set of pragmatic reasons for using Spin for verifying C code and the challenges that need to be overcome. Section 3 provides an overview over MPS and mbeddr, the technological base used to implement our approach. Section 4 presents our verification harness definition language with a set of language concepts needed for describing harnesses at a higher level of abstraction and their translation into Promela. In Section 5 we describe how our approach can be instantiated to verify various classes of SUVs that are often encountered in practice. Section 6 provides a discussion about the approach and results obtained so far, while Section 7 reviews the known body of related work. Section 8 concludes the paper and sketches directions for future work.

2. SPIN FOR C CODE VERIFICATION

In the following we describe our case for using Spin for C-level verification, then we describe usability challenges of using Spin by practitioners and our approach to tackle them.

2.1 Advantages of Spin

Despite big improvements of software verifiers in the last years [2], there are a set of pragmatic reasons which make the use of Spin for code verification appealing when compared with specialized software model checkers.

Language subset for SUVs. To our knowledge, existing software model checkers work only with a subset of the C/C++ language. That is, compiler specific extensions, assembler fragments or complex features of C/C++ are not supported. In contrast, Spin can be used to verify every code which can be compiled.

Use of libraries. When the SUV uses libraries, software model checkers require the user to model the observable behavior of these libraries. However, precise modeling is many times impossible (or impracticable) and thereby the analyses results are imprecise. Environment-driven code checking with Spin does not suffer from this drawback because it generates a verifier that contains the code of the SUV together with any libraries used in binary form.

Code that defies verification. There are cornerstone cases where C code model checkers have difficulties in analyzing the code, e.g. use of floating-point numbers, deep or complex loops. In many of these cases environment-driven code checking offers an alternative because regions of the input space can be exhaustively verified easily, provided the execution of SUV is fast enough.

Combining testing with model checking. As described in [8] Spin allows naturally to blend exhaustive model checking with testing, e.g. by generating some explicit input data randomly in a deliberate manner when exhaustive model checking over the whole range of input data is not feasible.

Semantic misalignments. There is always a possibility of semantic misalignments between how the C compiler and the model checker interpret the code. This is true especially when the code contains fragments which are not specified by the C standard. Due to the fact that Spin verifies the binary, these misalignments cannot happen.

2.2 Usability Challenges

The reasons cited above are our main motivation to enhance the mbeddr platform with support for environment-driven code checking using Spin. Nevertheless the approach still poses challenges in its applicability to practitioners due to the following reasons:

Modeling the verification harness. Users are required to have a deep understanding of Promela and its idioms. Small mistakes in the specification of the environment spans, for example, a smaller state space than expected reducing its usefulness. Furthermore, descriptions of relatively simple environments tend to be verbose. Practitioners simply shy away in front of this (perceived) complexity.

Lack of IDE support. We are currently not aware of IDEs which enable the definition of Promela models and their integration with C code of the system under verification. Without IDE support, most of the code checking work must be done manually, e.g. building the verifier.

Tracking the SUV state. If the SUV has internal state, it must be tracked in the environment model such that the observed SUV behavior become deterministic and verification results repeatable. Otherwise the user looses control of the SUV and Spin cannot faithfully decide about a fault (both false positives and false negatives are possible).

Understanding counterexamples. The error trail generated by Spin in case of a violated propoerty is—especially when C code is embedded in the Promela model—verbose and hard to understand. Additional lifting mechanisms are therefore required that relate the steps of the error trail to the language constructs used in the environment model.

2.3 Tackling the Challenges

In order to address these challenges we are specifying environment models in a higher-level description language that deals with the peculiarities of verification harnesses to support practitioners. The design of this DSL is driven by the following goals:

G1) Simple syntax for harness definition: The syntax must be simple and based on a few high-level constructs to ease learning of a new language. In addition, the constructs must be also sufficiently expressive such that a wide variety of C components can be verified. It should "feel like" C and serve both beginners and advanced users equally.

G2) Support for understanding the witness: The error trail generated by Spin shall be enriched with additional information about the SUV state such that comprehension of the counterexample is easier. This goal implies also that the error trail is provided in a format which can be parsed and displayed in an easy-to-understand manner.

3. MPS AND MBEDDR

In the following we give a brief overview of the technological basis which is provided by the language workbench MPS and its instantiation for embedded C code development mbeddr. Afterwards we discuss the integration of Spin into this technology stack.

3.1 JetBrain's Meta-Programming System

Our work relies on language engineering technologies, which refer to defining, extending and composing programming and domain-specific languages and their integrated development environments (IDEs). Language workbenches are tools that support efficient language engineering. The language workbench MPS[2] supports all aspects of the definition of DSLs such as abstract syntax, advanced editors, type systems, code generators and analyzers. It serves as the core technology for our implementation.

3.2 MBEDDR

The tool mbeddr is an open source technology stack for embedded C code development and verification. It provides incremental, modular and domain-specific extensions of C implemented on top of MPS. Figure 2 shows an overview over the mbeddr architecture [19]. In a nutshell, mbeddr offers support for three concerns in the development of embedded systems: the *implementation concern* contains engineering features to support the generation of executable code, the *analysis concern* contains features that enable code analyzers and integrated test tools, and the *process concern* (not presented in figure for brevety reasons) offers support for the development process and software lifecycle like definition of requirements and of product lines.

The work presented in this paper is an extension of mbeddr's analysis concern. The analysis concern integrates different external analyses tools. Besides the integration per se of a tool, the analysis concern offers DSLs for defining verification harnesses which are verification-tool specific. At higher levels, mbeddr users have the possibility to use the ANSI-C specification language (ACSL [1]) for defining contracts on functions – from the ACSL contracts users can generate C-level assertions or comments that tools like Frama-C can consume. Another possibility is that the users express more complex verification conditions through property patterns which are translated into C snippets containing assertions. Different high-level abstractions come with their specific analyses – e.g. completeness and consistency of decision tables [16]. The most advanced integration is that of CBMC [5]. The integration of the other tools are at different stages of maturity and we are continuously experimenting with different verification technologies to enhance mbeddr's analysis capabilities.

[2]https://www.jetbrains.com/mps/

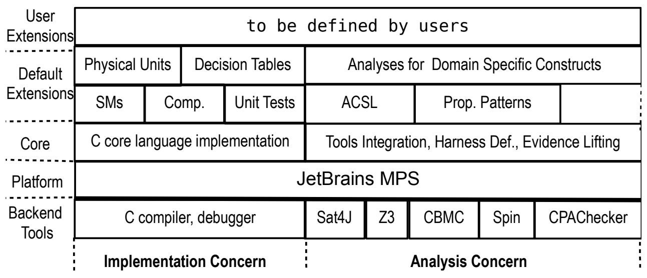


Figure 2: Overview over the technology stack of mbeddr; DSLs for coding (left-hand side), tools and DSLs for formal analyses (center).

3.3 Integrating Spin into MBEDDR

Language Definition. The work presented in this paper comprises the definition of DSLs for the design of verification harnesses, which starts with the design of the the language concepts and their relationships between them (abstract syntax). Figure 3 shows a high-level overview of how the Promela language implementation and its integration with other languages are supported in mbeddr. A PromelaModule contains entities of type IPromelaModuleContent which can be, for example, CDecl or ProcType statements. A CDecl statement contains top level elements of mbeddr's IModuleContent which enables the use of any language supported in mbeddr, C language in our case, to be used inside a Promela CDecl statement. As an extension of Promela language, we have defined the high-level harness DSL (Figure 3-bottom) as described in Section 4.

Generator and Build Process. Deploying features from the underlying MPS platform, we can generate text such that it can be passed to compilers or any other tools. The way forward to implement the Promela model generator for the verification harness description is to provide first a model-to-model transformation from the DSL for harness definition into the Promela language. Then, concrete code is generated from the Promela model (model-to-text transformation).

Once text is obtained, MPS allows the definition of build steps which contain the calling of Spin and GCC to compile the verifier pan. Once the pan executable is built, it is executed in a background process. If the verification fails, the produced error trail is read and the witness is fed back to mbeddr and displayed in its UI (see Figure 15-right).

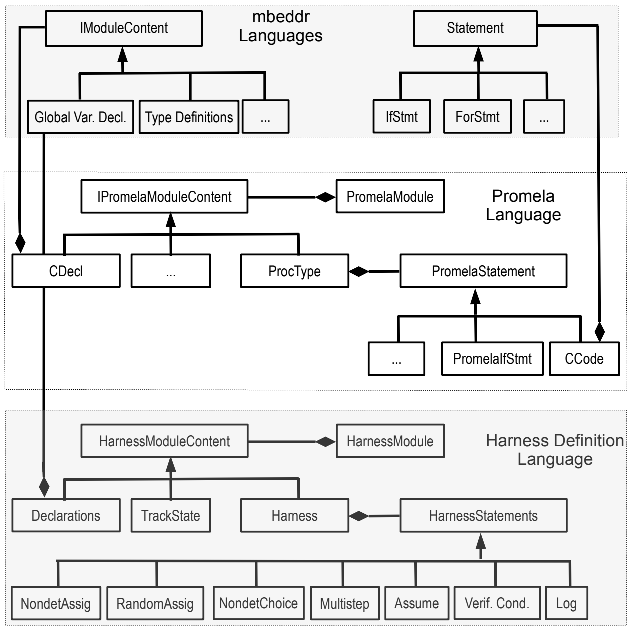


Figure 3: Overview of the language composition between mbeddr, Promela and the language for harness definition.

4. HARNESS DEFINITION LANGUAGE

We develop several high-level concepts of a harness definition language to enable a convenient definition of a verification harness. For each concept its rationale and its translation rules to Promela are described. Our aim is to make the harness look like C code with a minimal number of special constructs in order to lower the acceptance hurdle for developers who are the targeted users of the language. The language concepts are introduced "by example" to ease readability.

Harness Module Concept. A harness definition describing the environment, in which the SUV is executed, is realized in the *harness module* concept. It comprises the following four sections: 1) Import of C headers that describe the interface of the SUV used in the harness logic; 2) Declaration of input data used when calling the SUV; 3) Tracking of SUV state variables (optional); and 4) Specification of the harness logic, including assertions.

Figure 4 presents a harness skeleton and its encoding in Promela. The import section (line 1) contains a header file that describes the interface of the SUV. A variable declaration of an array of type tpe_t appears in line 3. Such declarations are later transformed to Promela c_decl statements in the build process. The next line contains the information about the tracked variables - this will be automatically translated into c_tracks with needed information about the size. All tracked variables are "UnMatched" and they do not go into the state vector. The harness logic is given in line 6. It carries a name and a body that contains the scenario of the call to the SUV which is detailed further below.

```
1  imports "suv.h"
2
3  decls { tpe_t[5] arr; }
4  track state: arr
5
6  harness logic_definition() { /* code */ }
```

```
1  c_decl {
2    \#include "suv.h"
3    tpe_t[5] arr;
4  }
5
6  c_track "&arr" "5*sizeof(tpe_t)" "UnMatched"
7
8  active proctype logic_definition() {
9    /* code */ skip
10 }
```

Figure 4: Encoding of the harness module.

Logging Concept. When an assertion fails, Spin prints a witness which contains the path taken through the Promela model of the verification harness and the values of its variables. Promela offers the Printf construct to print additional information. We leverage the Printf functionality and generate automatically Printf statements from all constructs that assign values nondeterministically or randomly. Furthermore, we allow users to log information in order to improve the readability of the witness. In order to recognize these statements later in the error trail, we use a special format for the strings printed—all strings denoting key/value pairs are prefixed with ###. Thereby we obtain a lifted witness representing a filtered error trail and presenting the relevant information to understand the failure at the harness level (Figure 15).

```
decls { int8_t var; }

harness logic_definition() {
  //code 1
  log witness(var);
  //code 2
}
```

```
c_decl { int8_t var; }

active proctype logic_definition() {
  //code 1
  c_code { Printf("### var=%d\n", var); }
  //code 2
}
```

Figure 5: Logging information in the witness.

Non-deterministic Assignment Concept. The statement nondet_assign assigns non-deterministically a value to a variable from a given range. The variables can have different types and the statement is adequately encoded into Promela depending on the variable type. The non-deterministic assignment concept is an extension of the select statement in Promela by dealing automatically with more data types (e.g. arrays with a constant size, enumerations) and ranges (e.g. ranges with discrete values). Furthermore and as discussed before, a Printf statement is added after each variable assignment to display the chosen value in a witness.

```
 1  decls { char ch; }
 2
 3  harness p() {
 4    // code 1
 5    nondet_assign(ch, {'a', '.', '\'});
 6    // code 2
 7  }
```

```
 1  c_decl { char ch; }
 2
 3  active proctype p() {
 4    // code 1
 5    byte tmp__ch;
 6    if
 7      :: tmp__ch = 'a';
 8      :: tmp__ch = '.';
 9      :: tmp__ch = '\\';
10    fi;
11    c_code {
12      ch = Pp->tmp__ch;
13      Printf("### ch = %c", ch);
14    }
15    // code 2
16  }
```

Figure 6: Encoding of nondet_assign where the domain is a set of discrete elements of type char. Besides the variable assignment itself, line 13 contains an automatically inserted Printf statement in the special witness format.

```
decls {
  enum EN {
    FIRST = 10, SECOND = 20, THIRD = 30
  } en;
}

harness p() {
  // code 1
  nondet_assign(en);
  // code 2
}
```

```
c_decl {
  enum EN {
    FIRST = 10, SECOND = 20, THIRD = 30
  } en;
}

active proctype p() {
  // code 1
  int en_idx;
  select(en_idx : 0 .. 2);
  c_code {
    if (Pp->en_idx == 0) {
      en = FIRST;
      Printf("### en = FIRST");
    }
    ...
  }
  // code 2
}
```

Figure 7: Encoding of nondet_assign for variables of an enumeration type.

Figure 6 illustrates this concept. While the value is non-deterministically assigned to the given variable, the backtracking feature of the model checker ensures that eventually all possible values are tried. Figure 7 shows the case when the variable is of an enumeration type, which is translated to Promela's select statement.

Random Assignment Concept. In many practical cases the range of the input values to the SUV is too big to be explored in an exhaustive manner. To deal with this situation the random_assign statement in the harness language assigns a given number of random values to a variable according to the range description and using a certain seed value—an approach presented originally in [8]. The seed value enables the replay of the verification run. Again, the random assignment concept is able to deal with variables of different types, see Figure 8.

Assumption Concept. Frequently the requirements for the SUV specify assumptions about the environment in form of constraints between valid inputs to the SUV. A proper environment definition thereby should enforce that the required constraints between inputs hold in order for them to be valid. For specifying such constraints, we define the assume statement that takes a Boolean expression representing the assumption. The assumption must hold for all subsequent statements executed. If the assumption does not hold, then the execution is broken, a backtracking step is performed and the exploration continues after this reached state. Figure 9 illustrates the concept and its translation to Promela

```
decls { long v; }

harness p() {
  // code 1
  random_assign(v, [200, 300], 2, 50);
  // code 2
}
```

```
c_decl { long var; }

active proctype p() {
  // code 1
  bool rndInit = false;
  int cnt = 0;
  if
    :: !rndInit ->
         rndInit = true;
         c_code { srand(2); }
    :: else -> skip;
  fi;
  do
    :: cnt < 50 -> cnt = cnt + 1;
    :: break;
  od;
  c_code {
    var = (rand() % (300 - 200)) + 200;
    Printf("### var = %d", var);
  }
  // code 2
}
```

Figure 8: Encoding of random_assign.

as a `goto` to a label automatically inserted at the end of the harness code.

```
harness p()
{
  // code 1
  assume(cond);
  // code 2
}
```

```
active proctype p() {
  // code 1
  if
    :: !cond -> goto end_label;
    :: else -> skip;
  fi;
  // code 2
end_label: skip;
}
```

Figure 9: Encoding of assume. If the assumed condition does not hold, we jump to the end of the process and force a backtracking step.

Non-deterministic Choice Concept. A possibility needs to be provided to select the next behavior of the harness logic to be executed in a non-deterministic manner. We support this via the `nondet_choice` construct which provides multiple guarded behaviors. A choice succeeds if the guard of the selected behavior is true. When none of the guards evaluate to true, then a default branch (always present) is executed, see Figure 10.

```
harness p()
{
  // code 1
  nondet_choice {
    choice: guard1 -> { /* code1 */ }
    choice: guard2 -> { /* code2 */ }
    default: { /* code3 */ }
  }
  // code 2
}
```

```
active proctype p() {
  // code 1
  if
    :: guard1 -> { /* code1 */ }
    :: guard2 -> { /* code2 */ }
    :: else ->   { /* code3 */ }
  fi;
  // code 2
}
```

Figure 10: Encoding of nondet_choice.

Multi-Step Verification Concept. When the SUV exhibits internal state we need multi-step verification. It assumes that the SUV is called in a loop with different input values for each call. To facilitate this behavior in the harness definition, we define the `multistep` statement that is translated in Promela to a loop. Additionally the iteration number is logged to ease understanding a witness (Figure 11).

```
1  harness p()
2  {
3   // init
4   multistep(N) {
5    // body
6   }
7  }
```

```
1  active proctype p() {
2    // init
3    int __crtStep=0;
4    do
5      :: __crtStep < N -> {
6        c_code {
7          Printf("### Iteration = %d\n",
8                                 __crtStep);
9        }
10       // body
11     }
12     :: else -> break;
13   od;
14 }
```

Figure 11: Encoding of multistep as a loop; also witness information about the current state is printed (line 8).

Further Concepts. Besides the concepts listed above, our harness definition language offers additional features such as calling the SUV inside the harness, variable assignments, loops, and assertions. The translation of these concepts into Promela is straightforward.

Whenever the generator from the harness DSL into Promela encounters references to C code, or C statements, it automatically wraps them with `c_expr` or `c_code`.

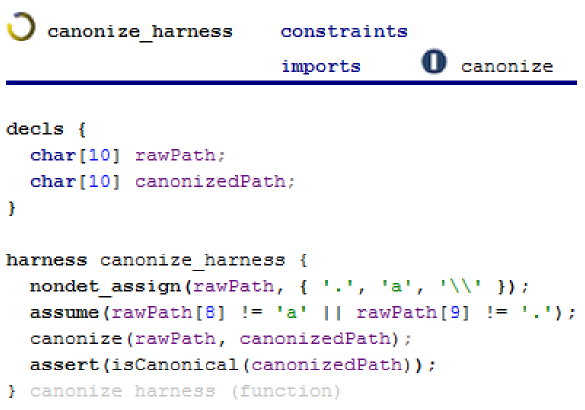
5. COVERED SUV CATEGORIES

In this section we present typical categories of SUVs that are subjected to the environment-driven code verification approach. Each of the sub-sections from below deals with one category and demonstrates how the verification harness can be modelled using the language defined in the previous section.

5.1 Cat I: Side-Effects Free Function

This category covers the case when the interface of the SUV is represented by a function which is side-effects free, i.e. its output depends solely on the provided input data. The set of input and output parameters is distinct. In this case, the harness is composed of variable declarations for inputs and outputs of the SUV and the main harness logic. Typically the harness logic produces non-deterministically values for inputs to the SUV and filters the valid ones using the `assume` statement. SUV outputs, i.e. return values, are compared with pre-calculated values in `assert` statements.

In Figure 12 we present an example of a harness for verifying the `canonize` function for path names (similar to the one presented in [11]) using our DSL. The harness generates all possible strings of 10 characters which contain '.', 'a' or '\', but do not end with the suffix "a.".



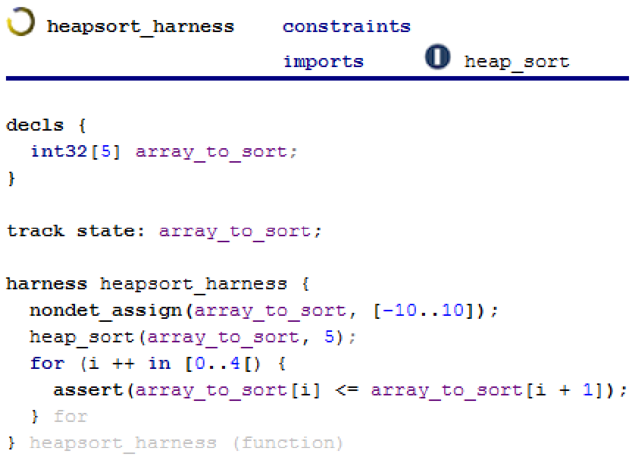
```
canonize_harness        constraints
                        imports       canonize

decls {
  char[10] rawPath;
  char[10] canonizedPath;
}

harness canonize_harness {
  nondet_assign(rawPath, { '.', 'a', '\\' });
  assume(rawPath[8] != 'a' || rawPath[9] != '.');
  canonize(rawPath, canonizedPath);
  assert(isCanonical(canonizedPath));
} canonize harness (function)
```

Figure 12: Example of harness definition for verifying a side-effects free function.

5.2 Cat II: In/Out Parameters

This category describes cases when the SUV interface is represented by a side-effects free function, of which some parameters are used both as inputs and as outputs. In addition to the harness definition of category I, the harness must track the variable which is used as an actual in/out parameter in the call of the function.

Figure 13 presents an example of a harness to verify a sorting algorithm `heap_sort` that sorts the elements of a given array "in place". At first, an array with five elements is non-deterministically initialized with values between -10 and 10, then the sorting function is called and the returned array is checked for correctness of the function. Due to the fact that the sorting happens in place, the state of the array needs to be tracked such that, when backtracking is performed, the search algorithm can continue from the last state stored in this array. In this example, backtracking happens each time after an assignment to the array was processed and the SUV is called again with the next assignment.

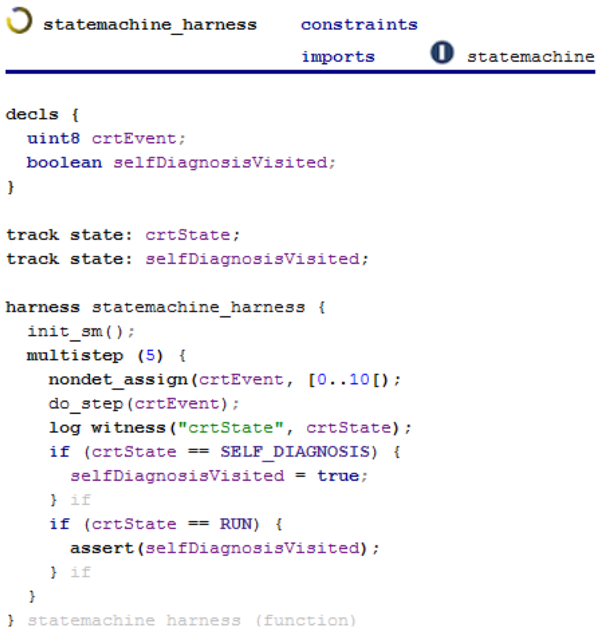


```
heapsort_harness        constraints
                        imports       heap_sort

decls {
  int32[5] array_to_sort;
}

track state: array_to_sort;

harness heapsort_harness {
  nondet_assign(array_to_sort, [-10..10]);
  heap_sort(array_to_sort, 5);
  for (i ++ in [0..4[) {
    assert(array_to_sort[i] <= array_to_sort[i + 1]);
  } for
} heapsort_harness (function)
```

Figure 13: Example of a harness definition for verifying a function with in/out parameters.

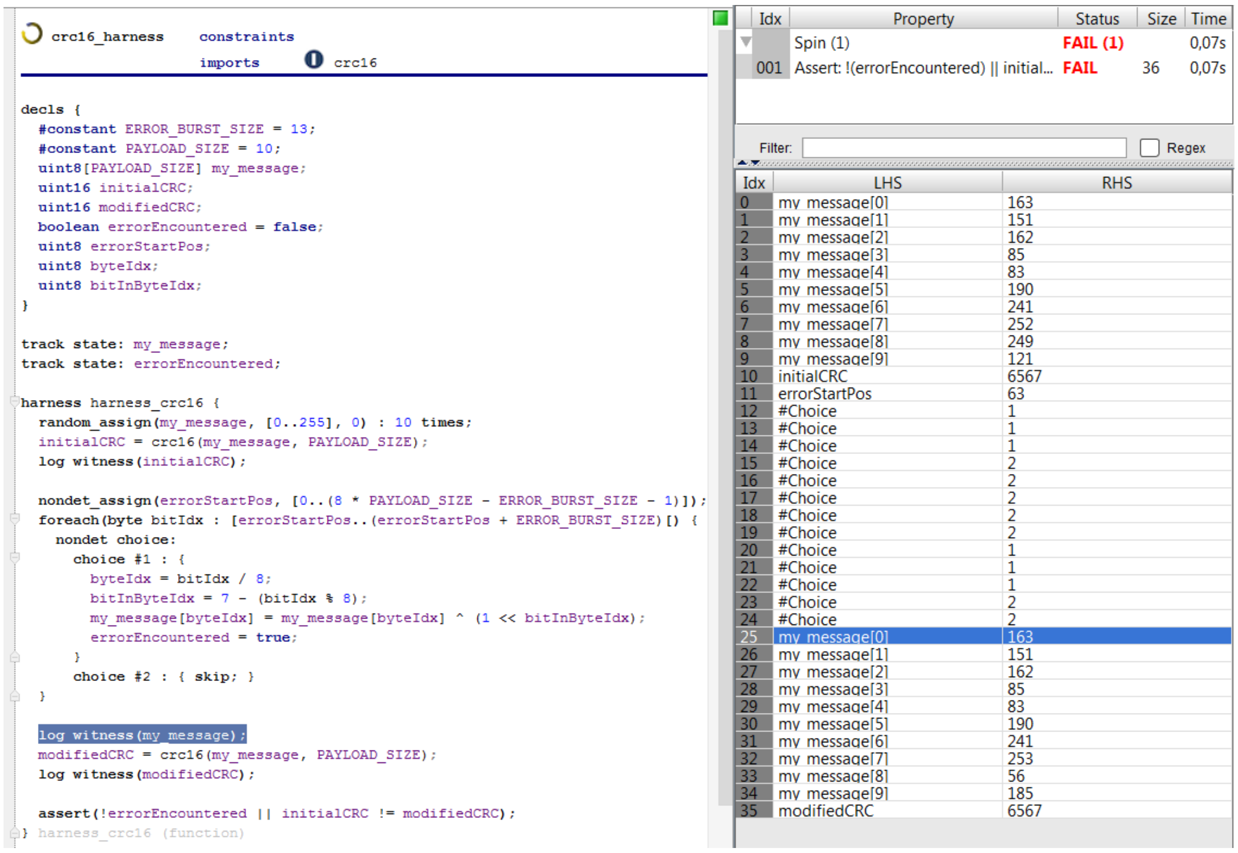
5.3 Cat III: State-based Verification

Another category exists when the SUV contains state and its behavior depends on that state. In this case a multi-step verification is required. A typical example for this category are state machines implemented in C. The state machine is typically implemented through a C function that processes an input parameter as input event that triggers the execution of the state machine.



```
statemachine_harness        constraints
                            imports       statemachine

decls {
  uint8 crtEvent;
  boolean selfDiagnosisVisited;
}

track state: crtState;
track state: selfDiagnosisVisited;

harness statemachine_harness {
  init_sm();
  multistep (5) {
    nondet_assign(crtEvent, [0..10[);
    do_step(crtEvent);
    log witness("crtState", crtState);
    if (crtState == SELF_DIAGNOSIS) {
      selfDiagnosisVisited = true;
    } if
    if (crtState == RUN) {
      assert(selfDiagnosisVisited);
    } if
  }
} statemachine harness (function)
```

Figure 14: Example of a harness definition for verifying a state machine.



```
crc16_harness    constraints
                 imports        crc16

decls {
  #constant ERROR_BURST_SIZE = 13;
  #constant PAYLOAD_SIZE = 10;
  uint8[PAYLOAD_SIZE] my_message;
  uint16 initialCRC;
  uint16 modifiedCRC;
  boolean errorEncountered = false;
  uint8 errorStartPos;
  uint8 byteIdx;
  uint8 bitInByteIdx;
}

track state: my_message;
track state: errorEncountered;

harness harness_crc16 {
  random_assign(my_message, [0..255], 0) : 10 times;
  initialCRC = crc16(my_message, PAYLOAD_SIZE);
  log witness(initialCRC);

  nondet_assign(errorStartPos, [0..(8 * PAYLOAD_SIZE - ERROR_BURST_SIZE - 1)]);
  foreach(byte bitIdx : [errorStartPos..(errorStartPos + ERROR_BURST_SIZE)[) {
    nondet choice:
      choice #1 : {
        byteIdx = bitIdx / 8;
        bitInByteIdx = 7 - (bitIdx % 8);
        my_message[byteIdx] = my_message[byteIdx] ^ (1 << bitInByteIdx);
        errorEncountered = true;
      }
      choice #2 : { skip; }
  }

  log witness(my_message);
  modifiedCRC = crc16(my_message, PAYLOAD_SIZE);
  log witness(modifiedCRC);

  assert(!errorEncountered || initialCRC != modifiedCRC);
} harness_crc16 (function)
```

Idx	Property	Status	Size	Time
	Spin (1)	FAIL (1)		0,07s
001	Assert: !(errorEncountered) || initial...	FAIL	36	0,07s

Filter: Regex

Idx	LHS	RHS
0	my_message[0]	163
1	my_message[1]	151
2	my_message[2]	162
3	my_message[3]	85
4	my_message[4]	83
5	my_message[5]	190
6	my_message[6]	241
7	my_message[7]	252
8	my_message[8]	249
9	my_message[9]	121
10	initialCRC	6567
11	errorStartPos	63
12	#Choice	1
13	#Choice	1
14	#Choice	1
15	#Choice	2
16	#Choice	2
17	#Choice	2
18	#Choice	2
19	#Choice	2
20	#Choice	1
21	#Choice	1
22	#Choice	1
23	#Choice	2
24	#Choice	2
25	my_message[0]	163
26	my_message[1]	151
27	my_message[2]	162
28	my_message[3]	85
29	my_message[4]	83
30	my_message[5]	190
31	my_message[6]	241
32	my_message[7]	253
33	my_message[8]	56
34	my_message[9]	185
35	modifiedCRC	6567

Figure 15: Harness definition for checking a CRC-16 algorithm (left) and the presentation of a witness (right). A double-click on a witness entry selects the corresponding node in the harness definition. Additional logging of harness variables is taken in the witness and facilitates its understanding.

In Figure 14 an example of a harness for verifying a state machine is presented. The SUV contains the function for initialization `init_sm()` and the `do_step()` function for stepping through the state machine. Because the harness needs to reason about the correct state after each step, all variables related to the SUV state should be tracked in the harness. The `multistep` statement ensures the continued execution of the state machine for the given number of steps.

5.4 Cat IV: Huge Input Spaces

We often encounter situations when the oracle has a complex structure and cannot be captured through simple assertions or temporal logic patterns. An example system for this category is an error-detection algorithm like the Cyclic Redundancy Check (CRC). Such algorithms might exhibit subtle defects (e.g. when a "bad" polynomial is used) or their implementation might be simply faulty. Testing CRC algorithms is difficult because of the huge input space caused by the multitude of possible messages and the occurrence of error bursts. In order to tackle the complexity of the input space, it is sensible to combine random testing with exhaustive verification. Parameters with an input space that is too large to be systematically explored are subjected to a random selection of values, while for the other input parameters exhaustive verification is applied.

In Figure 15-left a harness is presented for checking an implementation of the CRC-16 algorithm. The payload is initialized randomly (in our example 10 bytes in the array `my_message`) for 10 times. That is, 10 different, randomly chosen payloads are generated and submitted to the SUV which calculates the CRC-16 value. Finally, all possible burst errors up to length 13 are applied on the telegram at any location inside the payload. A chosen error burst is either applied or skipped; see the `nondet_choice` statement. The CRC-16 value for the disturbed payload is computed again. It is expected that the CRC-values for the original payload and the disturbed one are different under all considered cases, which is formulated in the corresponding `assert` statement. In order to print additional information in the counterexample we inserted additional `log witness` statements.

Figure 15-right presents a witness returned by Spin and then lifted in mbeddr. Due to the special format of the logging construct, the tool mbeddr-spin can establish the relation between the witness entry and the corresponding statement in the harness definition. By double-clicking on an entry of the witness, the corresponding harness part from which that witness entry originates will be selected in the editor. The link between the witness and the editor makes it easier to understand big witnesses and navigate theough the harness code.

6. DISCUSSION

The approach introduced in Sections 4 and 5 led to the development of the tool mbeddr-spin that extends the feature set of the mbeddr platform. It targets specifically the development of safety-critical embedded C code. In this application domain, environment-driven code verification has its advantage because it avoids any semantic misinterpretations of code between the compiler and the model checker that could render results obtained from direct C code verification useless (see Section 2). The mbeddr-spin tool is currently in its prototype stage where we conduct experiments with C code functions from a number of industrial projects to evaluate its usefulness. The evaluation covers (1) the applicability of the tool and (2) its usability as observed by users. First results are reported below.

6.1 On Practical Applicability

The use of Spin as the underlying model checker proofs beneficial due to its maturity. Environment models for SUVs at unit test level tend to describe little behavior and posses complexity rather in the definition of the input data domain.

Runtime performance of the pan executable is mostly constraint by the execution time of a called C function of the SUV. Performing myriad calls of the SUV quickly exhausts the available time resources. Thus, the total number of SUV calls needs to be controlled by the engineer. Consider the verification of the CRC-16 algorithm (Figure 15). The verification harness applies 10 different payloads, each 10 bytes long. The generation of mutated payloads with error bursts up to length 13 produces $(8 * 10 - 13) * 2^{13} = 548864$ modified payloads. On modern machines Spin is able to cover this amount of data in a few seconds.

A limitation of the approach applies when verifying state-based SUVs whose state variables are (partly) not accessible (e.g. when the SUV uses a library that has internal state). In this case the backtracking functionality of Spin cannot be used because the state of the SUV cannot be backtracked. One could rely on the availability of a reset function of the SUV that brings it back to its initial state. This assumption is frequently met in practice. However a complete SUV reset as the only possibility for backtracking requires a modified state exploration strategy inside the model checker: when backtracking, the model checker re-starts from the initial state of the model and performs the (shortest) path to reach the backtracked state, from where it chooses another execution path of the model and, hence, in the SUV.

6.2 Revisiting design goals G1 and G2

We conducted the work in order to support engineers in the application of formal methods by hiding them under the hood of a proper language and IDE. The use of a DSLs stack such as mbeddr is very helpful in this respect. It allows the extension of the IDE that the engineer already knows with additional features. For example, syntax highlighting and auto-completion are provided out of the box for the newly designed harness definition language. Such usability features make the application of a new language easy and less laborious as well as attractive and motivating to learn.

Moreover, we succeeded in the design of a homogeneous language that gets rid of the intertwined usage of Promela and C code (goal **G1**). The offered language concepts for harness definitions are at a higher level of abstraction than Promela such that a common language integrating both Promela and C could be defined.

Deploying the analysis aspect of mbeddr we were able to implement debugging support based on the witness filtered from a Spin error trail (goal **G2**). The user only sees entries in the witness that have a direct correspondence to elements of his harness definition. Details from the underlying Promela model are hidden to him. This way, the user can click on an entry from the witness and the corresponding harness code is highlighted (see Figure 15).

7. RELATED WORK

The paper [12, 11] introduces the concept of model-driven code verification using Spin. In this approach, a Promela model represents the environment, in which the code to be verified is embedded and which guide the verification process. The work in [8] presents a unified framework which uses Promela both for test harness definition and model checking. Our work directly builds on these approaches and extends them by defining a higher-level harness definition language which captures commonly encountered patterns of harness definitions. Moreover, we deeply integrate Spin in the mbeddr development environment and thereby offer the same IDE for writing C code and and harness definitions using the new DSL. For this reason the whole approach becomes easier to understand and use by software development practitioners. Furthermore, advanced users can write complex harnesses easier.

The paper [18] presents an approach which uses Spin to check the C code generated from a high-level DSL for describing rules. The verification harness is automatically generated and the verification result and witness is lifted back at the level of the DSL. Compared to this work our approach allows the verification of SUVs written in generic C code or DSLs of mbeddr.

Similarly, the work in [9, 10] introduces a DSL for describing harnesses for testing. This DSL abstracts away from often encountered idioms in order to increase the readability of state-space descriptions. The DSL descriptions allow users to embed code fragments exercising the system under test in its host language. Subsequently test scripts are generated in the target language (Python or Java). Compared to this work, our focus is clearly on leveraging Promela/Spin capabilities for C code verification; we also offer full IDE support for editing verifying and interpreting verification results.

There has been already done work for integrating Promela into common IDEs. For example, the papers [3] and [6] present Eclipse-based IDEs for Promela. Their integrations support Promela users with modern editing facilities like code completion, syntax highlighting, and on-the-fly type checks. Furthermore, language engineering technologies such as Xtext are deployed. While the Eclipse IDE integrations offer modern tool support for building Promela models, none of these contributions support the capability of integrating C code in Promela. Furthermore, the editors are essentially tied to Promela and do not allow users to use higher-level abstractions for capturing common idioms.

Other body of work deals with Promela extensions. The paper [17] presents several light-weighted patterns for using Promela. These patterns are candidates to be lifted at a higher-level DSL. The work in [7] presents an extension of Promela with stronger type checks. While the paper [13] apply language engineering technologies to implement new language features which can be seamlessly added to Promela. mbeddr-spin uses the same kind of approach for leveraging model-driven code checking with Spin.

8. CONCLUSIONS

The paper presents an approach to improve the usability of model-driven C code verification using Spin. It proposes a verification harness definition language that is built on top of Promela. For this purpose, language engineering technologies of the language workbench MPS are used. The new language is introduced by describing the most important language concepts that wrap Promela and its embedded C code. The language captures common usage patterns for defining verification harnesses. The applicability of the language is shown by discussing several categories of SUVs that are often encountered in practice. Compared with the original approach of building environment models that relies on the native usage of Promela and embedded C code, the new language approach simplifies the harness definition considerably.

Our own future work goes along the following directions. Firstly, we aim to pilot the approach in industrial projects within Siemens to gain further insights in the harness definition for a wider range of C implementations and try to identify further definition patterns and optimized language encodings in Promela. Secondly, we plan to disseminate this technology and train the engineers to perform the verification themselves as a much enhanced (black-box) unit test. Thirdly, we want to address the verification of a SUV with inaccessible internal state. This work is part of a larger effort to bring formal verification technology closer to practitioners. We think that the best ultimate solution is to integrate functional verification in the build workflow of *continuous integration* for software, similarly to the integration of automated testing.

9. REFERENCES

[1] P. Baudin, P. Cuoq, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. ACSL 1.4 : ANSI/ISO C specification language. Technical report, CEA LIST, 2010.

[2] D. Beyer. Reliable and reproducible competition results with benchexec and witnesses report on SV-COMP 2016. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2016.

[3] Z. Brezocnik, B. Vlaovic, and A. Vreze. SpinRCP: the Eclipse rich client platform integrated development environment for the Spin model checker. In *International Symposium on Model Checking of Software (SPIN)*, 2014.

[4] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Operating Systems Design and Implementation (OSDI)*, 2008.

[5] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2004.

[6] B. de Vos, L. C. L. Kats, and C. Pronk. EpiSpin: An Eclipse plug-in for Promela/Spin using Spoofax. In *International Workshop on Model Checking Software (SPIN)*, 2011.

[7] A. F. Donaldson and S. J. Gay. ETCH: An enhanced type checking tool for Promela. In *Workshop on Model Checking Software (SPIN)*. Springer, 2005.

[8] A. Groce and R. Joshi. Random testing and model checking: Building a common framework for nondeterministic exploration. In *International Workshop on Dynamic Analysis (WODA)*, 2008.

[9] A. Groce and J. Pinto. A little language for testing. In *NASA Formal Methods (NFM)*, 2015.

[10] J. Holmes, A. Groce, J. Pinto, P. Mittal, P. Azimi, K. Kellar, and J. O'Brien. Tstl: the template scripting testing language. *International Journal on Software Tools for Technology Transfer*, pages 1–22, 2016.

[11] G. Holzmann, R. Joshi, and A. Groce. Model driven code checking. *Automated Software Engineering*, 2008.

[12] G. J. Holzmann and R. Joshi. Model-driven software verification. In *Workshop on Model Checking Software (SPIN)*. Springer, 2004.

[13] Y. Mali and E. V. Wyk. Building extensible specifications and implementations of Promela with AbleP. In *Workshop on Model Checking Software (SPIN)*, 2011.

[14] Z. Molotnikov, M. Völter, and D. Ratiu. Automated Domain-Specific C Verification with mbeddr. In *International Conference on Automatic Software Engineering (ASE)*, 2014.

[15] D. Ratiu, M. Voelter, B. Kolb, and B. Schätz. Using language engineering to lift languages and analyses at the domain level. In *NASA Formal Methods Symposium (NFM)*, 2013.

[16] D. Ratiu, M. Voelter, B. Schaetz, and B. Kolb. Language engineering as enabler for incrementally defined formal analyses. In *Proceedings of the Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches (FORMSERA'2012)*, 2012.

[17] T. C. Ruys. Low-fat recipes for spin. In *International Workshop on SPIN Model Checking and Software Verification*. Springer, 2000.

[18] M. Sulzmann and A. Zechner. Model checking dsl-generated C source code. In *International Workshop on Model Checking Software (SPIN)*, 2012.

[19] M. Voelter, D. Ratiu, B. Kolb, and B. Schätz. mbeddr: instantiating a language workbench in the embedded software domain. *Automated Software Engineering*, 2013.