



An integrated environment for Spin-based C code checking

Towards bringing model-driven code checking closer to practitioners

Daniel Ratiu¹ · Andreas Ulrich¹

Published online: 18 February 2019
© Springer-Verlag GmbH Germany, part of Springer Nature 2019

Abstract

Model-driven code checking (MDCC) has been successfully used for the verification of functional requirements of C code. An environment model that describes the context, which a program is expected to run in, is defined in Promela, translated to a model checker program by Spin, and linked with the program acting as system under verification. In this article, we summarise the practical advantages of MDCC which motivate its use in an industrial setting and discuss the challenges to its broader adoption. Environment models exhibit heavily intertwined Promela and C code statements, which make them hard to write and understand. We propose a high-level language for verification harness definition which hides the Spin engine under the hood. A small number of language concepts is sufficient to define verification harnesses for commonly encountered C programs. Widening the scope of the approach, we provide means to verify programs that exhibit internal state and extend the set of checked properties beyond classical assertions to those checked by LLVM/Clang code sanitizers. Thus, a user can focus on finding the best solution to combine exhaustive exploration of the environment with testing strategies. Our approach is prototypically integrated into mbeddr development platform. We present its instantiation on real-world code examples and discuss our experiences gained with the verification of software from the railway domain.

Keywords Domain-specific languages · Model checking · Testing · Code verification · Code sanitizers

1 Introduction

Formal verification increases the quality of software by enabling developers to discover subtle bugs. Despite dramatic improvements of the verification algorithms and tooling in the last decade, software verification tools are regarded as expert tools and are seldom used in practice. Rather, developers prefer automated checks on generalisable software properties like those related to code robustness (e.g. absence of undefined behaviour), potential security issues or the adequate use of libraries (e.g. memory allocation). The rise of static code analysis tools that deploy formal methods internally such as Polyspace [23] and Coverity [21] is evidence of this trend. Their appeal likely stems from the fact that these tools promise a push button approach for automating com-

mon development tasks that would require tedious manual code reviews otherwise.

Despite these gains, specific checks are still required to validate the correct implementation of requirements. To this day, the mostly deployed approach to software validation is testing. Its major advantage is that testing is applied directly on the executable software, which is close to the way it is finally used. Moreover, it limits greatly the occurrence of false positives, i.e. detected failures that do not describe actual software faults, which frequently hampers static software analysis tools. On the other hand, the disadvantage of testing lies in the efforts to specify and implement a test driver that sufficiently covers the practically infinite input space of industrial software systems.

Addressing the question on sufficient coverage in testing, an alternative approach to ever more sophisticated test methods is to restrict the state space, in which the system is intended to run, to a meaningful size in the first place, and cover it systematically and, possibly, exhaustively in the tests afterwards. This approach requires to define an *environment model* of the system, i.e. the context, in which the

✉ Daniel Ratiu
daniel.ratiu@siemens.com

Andreas Ulrich
andreas.ulrich@siemens.com

¹ Siemens, Corporate Technology, Munich, Germany

system will be deployed. To guarantee exhaustive coverage of the environment model, one can rely on the power of model checking for its exploration and deriving tests. This approach was pioneered in the paper by Holzmann and colleagues [15,16]. It applies Spin¹, a highly mature explicit state model checker, on an environment model specified in Promela [14] that calls the *Software Under Verification* (SUV) directly via embedded calls in C code. Spin generates a verifier that systematically explores the states of the environment model and calls the SUV in the different states of this model. The approach was extended further by Groce and Joshi [11] by incorporating random testing and blending it with model checking. This extension enables dealing with large input spaces that are infeasible to cover systematically by model checking due to state explosion. The developer has a full range of possibilities at hand for describing environments: from a trivial model with no constraints on the SUV input (the most general model) to a highly elaborated model with precisely constrained data sets. This flexibility is often required in practice to deal with a multitude of different software systems that need to be validated under stringent resource constraints.

This article sets forth previous work on model-driven code checking [11,15,27] along the following directions:

Harness definition language

- We develop a *verification harness definition language* to raise the level of abstraction for specifying environment models. Our language contains a small number of constructs which are enough to encode the harness for a broad category of systems.
- We demonstrate how these language concepts are encoded in Promela to derive executable environment models in Spin and how to check SUVs with hidden internal state;

Integrated environment

- We develop a way to *express witnesses of failed verification runs to ease their understanding* in the context of the environment model.
- We add the possibility to check for *robustness properties* of SUV code by using the sanitizers provided by LLVM [18];
- We show how the *language workbench* MPS² enabled us to prototypically extend mbeddr [36], an integrated development environment for C code, with support for MDCC and develop mbeddr-spin³.

¹ <http://spinroot.com/spin/whatispin.html>.

² <https://www.jetbrains.com/mps/>.

³ <https://sites.google.com/site/ratiud/tools>.

Applicability in practice

- Our work is motivated by practical industrial needs to check C components as detailed in Sect. 3.
- We discuss how our approach can be used to check several categories of real-world C components;
- We report our initial experience from the application of our approach in an industrial setting.

Outline

First, an overview of model-driven code checking and our enhancements are presented and their advantages but also challenges to their practical use are discussed. In addition, the base technologies MPS and mbeddr used to implement our approach are briefly introduced (Sects. 2–4). In the following, we introduce our verification harness definition language and the integration of code sanitizers (Sects. 5 and 6). Afterwards, the application of the language is shown on real-world C code components (Sect. 7) and we present preliminary experiences with an industrial application (Sect. 8). Finally, we discuss variation points, the related work and conclude the paper (Sects. 9–11).

2 Overview of the chosen approach

2.1 Model-driven code checking (MDCC)

MDCC [15,16] is a technique for checking software components written in C against an environment model using Spin. The *environment model* is provided by the user and formalises a part of a functional requirement. It defines the context in which the software component will be deployed. Originally, the approach uses Promela, the input language of Spin, for describing the environment.

The environment model, augmented with additional verification conditions, is called the *Verification harness* throughout this article. The harness is processed using Spin to generate the model checker program that explores it. The C program, called *System Under Verification* (SUV), interacts with the verification harness via SUV function calls, written as C code fragments embedded into Promela, invoked in the harness. Both, verification harness and SUV form a closed-loop system that evolves during the execution of the generated model checker. Inputs to the SUV are performed by executing calls to SUV functions, while outputs from the SUV, which comprise return values or tracked global SUV variables, are verified against conditions written directly in Promela as asserts or against embedded C code snippets acting as monitors.

A prerequisite of this approach is that calls to the SUV are *atomic*, i.e. there are no extra states in the verification

harness or SUV that are untracked by Spin [15]. Systematic state exploration relies on backtracking from the current state to a state that was visited before but still has unexplored behaviour. These states must be solely under control of the model checker.

2.2 Supportive language engineering

Deploying Spin for the exploration of the verification harness 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 for this purpose. Moreover, the manual effort required to produce a correct verification harness containing embedded C code fragments in an idiomatic manner can be big. To make formal verification techniques more usable for practitioners, we use language engineering technologies to hide the modelling complexity through the definition of a *Domain-Specific Language* (DSL) at a higher level of abstraction [25,28]. The approach is prototypically implemented in *mbeddr* [36], an *Integrated Development Environment* (IDE) and open source stack of DSLs on top of C for the development of embedded software.

Outlining our language engineering approach, consider Fig. 1 that sketches an example of a harness for checking a sorting algorithm `sort()`. The verification harness `sort_harness()` is provided as a Promela model (bottom). 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 13–16). The harness requires a mixed usage of Promela statements and embedded C code fragments in an idiomatic manner. The top part of Fig. 1 illustrates the proposed *Verification Harness Definition Language* which hides the repetitive parts in the Promela model and offers higher-level abstractions like *nondet_assign*.

3 MDCC opportunities and challenges

Our work is motivated by practical concerns with deploying the verification technology to a broader spectrum of software developers. In Sect. 3.1, we highlight practical advantages of the MDCC approach over static software model checking approaches. However, these advantages come with the challenge for the supporting tools as discussed in Sect. 3.2. We end with a set of design goals for tooling to address these challenges.

3.1 Advantages of model-driven code checking

Despite big improvements in software verification in the last decade [2], there are a number of pragmatic reasons which

```

1 decls { uchar my_array[4], *res; }
2
3 harness sort_harness {
4   nondet_assign(my_array, 0, 255);
5   res = sort(my_array, 4);
6   for(i = 0; i < 3; i++) {
7     assert(res[i] <= res[i+1]);
8   }
9 }

```

```

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    } } }

```

Fig. 1 Comparing the verification harness definition using the proposed higher-level DSL (top) with the idiomatic description using Promela and C code directly (bottom)

make MDCC appealing for the verification of C code. These reasons, detailed below, hold even against the advantages that modern software model checking tools like CBMC [7] or CPAchecker [3] offer.

3.1.1 C/C++ language support

To our knowledge, existing software model checkers work only with subsets of the C/C++ language. Compiler-specific extensions, assembler fragments or complex language features are not supported. In contrast, MDCC can be used to verify every code that can be compiled.

3.1.2 Use of libraries

When a SUV uses libraries, software model checkers require to model the observable behaviour of these libraries as well. However, precise modelling is many times impossible or not practical resulting into imprecise analyses results. MDCC does not suffer from this drawback because it generates a verifier which simply calls the SUV and its libraries.

3.1.3 Code that defies verification

There exist cornerstone cases that software model checkers struggle with when analysing the code, e.g. when the SUV uses complex computations with floating-point num-

bers, dynamically allocated memory or complex and deep loops. In many of these cases, MDCC offers an alternative because parts of the input space can be explicitly modelled and then exhaustively verified.

3.1.4 Semantic misalignments

There is always a chance that the interpretation of code differs between the compiler and the model checker. This is true especially when the language, such as C, exists in many compiler-specific variants. Due to the fact that MDCC verifies the compiled binary, these misalignments cannot happen.

3.1.5 Leverage of compiler-based sanitizers

Modern compilers provide *code sanitizers* [22,30,31] to check code for various failures at runtime, e.g. undefined behaviours or bad memory addresses. Sanitizers instrument the code with automated checks executed at runtime which can be combined with the MDCC approach.

3.1.6 Combining testing with model checking

Spin allows to blend exhaustive model checking with testing [11]. A user can decide to generate some input data randomly and explore other parts of the environment exhaustively. Compared to conventional testing, MDCC offers advantages that stem mainly from its integration of model checking and enables exhaustive coverage of selected aspects of a system requirement.

3.2 Challenges of model-driven code checking

The reasons cited above represent a strong case for using MDCC for the verification of C programs. However further efforts are required to make the approach applicable to practitioners in the industry. The following hurdles need to be overcome:

3.2.1 Modelling of the verification harness

Developers are required to have a deep understanding of Promela and its idioms. Small mistakes in the specification of the environment model could result into a state space with limited usefulness. Furthermore, descriptions of relatively simple environments tend to be verbose and exhibit intertwined statements of Promela and C code.

3.2.2 Understanding counterexamples

The error trail of a verification run which is produced in case of a failure is—especially when C code is embedded in the Promela model—verbose and hard to understand. Additional

mechanisms are required to relate the steps of the error trail to the statements of the harness.

3.2.3 Dealing with SUV internal state

If the SUV has internal state, it must be tracked in the verification harness such that the observed behaviour becomes deterministic and verification results repeatable. Therefore support for dealing with internal state (both visible and invisible to the harness) is indispensable.

3.2.4 Checking code robustness properties

Functional properties can be expressed using powerful temporal logic in assertions. However, a big class of properties are related to program robustness (e.g. overflows, bad memory access) and are covered by code sanitizers. To leverage their use the failure reports they produce need to be played back and coordinated with Spin's error trails.

3.2.5 Lack of IDE support to combine Promela and C

We are currently unaware of modern IDEs that facilitate the definition of Promela models and their integration with C code and provides syntax highlighting, auto-completion or consistency checking.

3.3 Tackling the challenges

In order to address these challenges, we develop a verification harness definition language that is capable of expressing an environment model at a high-level of abstraction. The design of this language and its supporting tooling is driven by the following goals:

3.3.1 Simple syntax for harness definition

The syntax must be simple and based on a few concepts to ease learning the new language. In addition, the concepts must be sufficiently expressive such that a wide variety of C programs can be verified. It should “feel like” C and look familiar to non experts in formal verification.

3.3.2 Support for understanding the witness

The error trail generated by Spin shall be enriched with additional information about the input vectors such that comprehension of the counterexample is easier. It should be easy to use the witness as test vector such that subsequent code debugging can be immediately performed. Furthermore, tooling is needed to relate witness entries to the parts of the harness from which they originate.

3.3.3 Support for dealing with internal state of SUV

It should be easy to deal with accessible SUV state and possible to deal with cases when the SUV does not have accessible internal state.

3.3.4 Integration with code sanitizers

Supporting sanitizers requires an integration of their error reporting mechanism with the error trail generation in Spin to ease the understanding of an observed failure when the verification is performed.

4 Technological base: MPS and MBEDDR

This section gives a brief overview of the technologies we use: the language workbench MPS and the development platform mbeddr. After this, we provide a brief presentation of the integration of MDCC into mbeddr.

4.1 JetBrains' meta-programming system

The presented work relies on language engineering technologies which cover the definition, extension and composition of domain-specific languages (DSLs) and their integrated development environments (IDEs). The language workbench MPS supports all aspects of the definition of DSLs such as abstract syntax, advanced editors, type systems and code generators. A detailed presentation of language engineering topics and case studies about systems built with MPS are contained in [6,34,35]. The following paragraphs focus on the main features of MPS relevant for our implementation.

4.1.1 Language extension and composition

DSLs are based on an abstract syntax that can be compared to a regular object-oriented design. MPS offers a meta-modelling approach to deal with abstract syntaxes which enables a user to introduce new concepts for existing languages and to design entirely new languages. At the core of MPS lies a so-called *projectional editor*, which renders, or projects, a program's abstract syntax in a concrete representation format. Working with a projectional editor avoids typical challenges of language design using a grammar (e.g. resolving ambiguities). It is the base for the development modular and extensible languages.

Our own implementation of the Promela language is based on the existing DSLs provided by mbeddr which is built using MPS. Additionally, we designed the verification harness definition language that sits on top of Promela, as shown in Fig. 3 and detailed in Sect. 5.

4.1.2 Rich context sensitive constraints

MPS offers different mechanisms for the definition of context sensitive constraints: scoping rules, type system checks or arbitrary checks written in Java. Some constraints prevent the construction of wrong models up-front; others lead to errors in the editor when they are violated. All these constraints are essential mechanisms to increase the usability of the tool by guiding the user towards the creation of correct models.

We have extensively used the constraints mechanism of MPS in order to implement editing guidance for our harness definition language, including support for its type system and scoping rules.

4.1.3 Modular and stackable model transformations

MPS offers two kinds of transformations: model-to-model transformations between instances of two different abstract syntaxes (the models) and text generation from a model. Implementing our approach requires mostly a transformation of a model given in one language into a model of another language; only at the very end of the transformation chain a text generator is used to output a text representation used for downstream compilation afterwards. Because the transformations convert typically an abstract model to a more detailed model, DSLs and their model-to-model generators form a stack.

Our stack of language extensions for defining verification harnesses provides such a stack: high-level verification harness concepts are reduced into lower level ones until Promela and C code can be generated as text.

4.2 MBEDDR

The tool mbeddr [36] is an open source technology stack for embedded C code development and verification and is built using MPS. Figure 2 shows a conceptual overview of the mbeddr modular DSLs build on top of each other and hosted by MPS. In a nutshell, mbeddr offers support for three development concerns:

- *Implementation* support for writing high-level DSLs and generating executable code, makefiles, etc.;
- *Analysis* support the definition of verification harnesses and properties, interaction with external analysers such as CBMC or Sat4J;
- *Process* support for process concerns such as the definition of requirements (not presented in the figure for brevity).

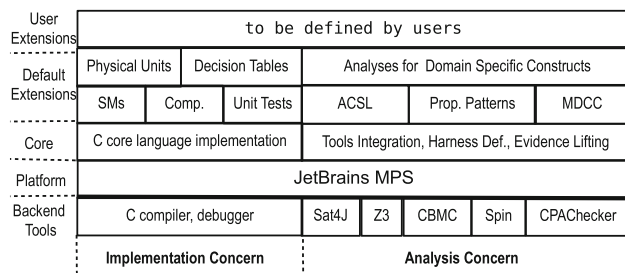


Fig. 2 Overview of the technology stack of mbeddr. DSLs are displayed for coding (left) and formal analyses (centre), among others. The DSLs are hosted by MPS and external tools like compilers or analysers are used in the backend

Our tooling, mbeddr-spin,⁴ is a prototypical extension of the analysis concern with support for MDCC.

4.3 Extending MBEDDR with MDCC

4.3.1 Language definition

Language concepts that are already present in mbeddr are extended with the concepts for Promela using the language engineering features of MPS. Figure 3 shows the integration of the Promela language implementation with other mbeddr languages using a simplified class diagram which contains only language concepts and the inheritance and composition relations. A *PromelaModule* contains entities of type *IPromelaModuleContent* like for example *CDecl* or *ProcType*. A *CDecl* contains top-level elements of mbeddr's *IModuleContent* like global variables. Additionally, the harness definition language, described in Section 5, is defined as an extension of the Promela language.

4.3.2 Generator and build process

We implemented model-to-model generators for the verification harness definition language and text generators for Promela models.

Furthermore, we implemented build steps for calling Spin with the generated source code to generate the `pan` program code and then the `gcc` or `clang` compiler to produce the executable `pan` program. If the model checker is built successfully, it is executed automatically in a background process. If the verification run fails, the produced error trail is read by MPS and the witness is lifted and displayed in the IDE (see Fig. 20).

⁴ <https://sites.google.com/site/ratiud/tools>.

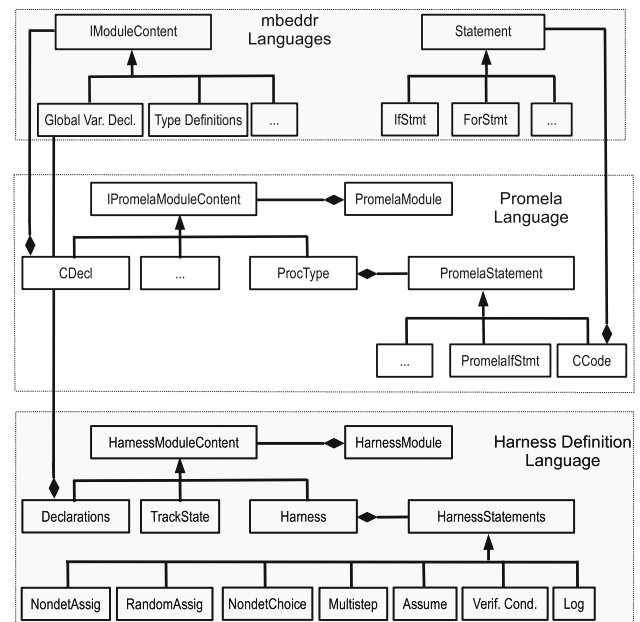


Fig. 3 Language composition in mbeddr with Promela and the harness definition language

5 Verification harness definition language

In this section, we introduce our language for defining verification harnesses. In Sect. 5.1, we present the core concepts of our language which are directly translated to Promela. Advanced concepts built on top of the basic ones are introduced in Sect. 5.2, while Sect. 5.3 is focused on verifying systems containing internal state.

5.1 Basic harness definition language

For each concept of our DSL, we describe its rationale and its translation to Promela. Our aim is to make the harness “look like” C code with a minimal number of special constructs in order to increase the acceptance by software developers. The language concepts are introduced “by example” to ease readability.

5.1.1 Harness module concept

A harness definition describes the environment, in which the SUV is executed. It is realised by the *harness module* concept which comprises the following four sections:

1. Import of C headers (`imports` keyword) that describe the interface of the SUV;
2. Declaration of input data (`decls`) used when calling the SUV from the harness;
3. Tracking of SUV state variables (`track state`); and

```

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)"
7                                     "UnMatched"
8
9 active proctype logic_definition() {
10   /* code */ skip
11 }

```

Fig. 4 A harness module is encoded as proctype. Top-level variables declarations are encoded into `c_decl` and tracked state into `c_track`

4. Specification of the harness logic (harness), 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 translated to Promela's `c_decl` statements. The next line contains information about a tracked variable; this will be translated to a `c_track` statement including information about the variable size. All tracked variables are automatically marked to be *UnMatched*, in accordance with [16], which implies that their values are stored in Spin's search stack, but they are not part of the state descriptor. The effect of tracking an unmatched variable is therefore that its previous value is restored when Spin performs a backtrack operation to a state already explored before. But the variable has no impact on the explored state space itself.

The harness logic is given in line 6. It carries a name and a body that contains the scenarios of calling the SUV, defined by using further language concepts.

5.1.2 Logging concept

When an assertion fails, Spin prints a witness that contains the path taken through the Promela model of the verification harness and the values of its variables, also known as error trail. Promela offers the `Printf()` function to log additional information. We leverage this functionality and generate automatically `Printf` calls from all constructs that assign values non-deterministically or randomly. Furthermore, we allow users to log additional information in order to improve the readability of a witness (`log witness` keyword). In order to recognise these log outputs later in the error

```

1 decls { int8_t var; }
2
3 harness logic_definition() {
4   //code 1
5   log witness(var);
6   //code 2
7 }

1 c_decl { int8_t var; }
2
3 active proctype logic_definition() {
4   //code 1
5   c_code { Printf("### var=%d\n", var); }
6   //code 2
7 }

```

Fig. 5 Logging information in the witness by means of the `Printf()` function provided by Spin. We log all messages using a special format which allows us parsing and reconstruction of the error trail

trail, we use a special format for the strings being printed. All strings denoting key/value pairs are prefixed with `###`, followed by the variable name and its value assigned by Spin (Fig. 5). A generated error trail can then be filtered for such prefixed log statements, which enables us to present the relevant information at the harness level (see Fig. 20).

5.1.3 Non-deterministic assignment concept

Using this concept, one can assign non-deterministically a value from a given range to a variable (`nondet_assign` keyword). 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, doubles) and ranges (e.g. ranges with discrete values). Furthermore, a `Printf()` function is added after each variable assignment to display the chosen value in a witness.

Figure 6 illustrates this concept. While the index of the value is non-deterministically assigned to the given variable, the backtracking feature of the model checker ensures that eventually all possible values are explored.

5.1.4 Random assignment concept

In many practical cases the range of 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 definition language assigns a given number of random values to a variable according to the range description and a certain seed value—an approach presented originally in [11]. The seed value enables the replay of the verification run. After the colon, we specify how often the variable is randomly assigned. The example from Fig. 7 presents the translation of 50 random assignments of the `long` variable

```

1 decls { char ch; double dbl; }
2
3 harness p() {
4   // code 1
5   nondet_assign(ch, {'a', '.', '\\'});
6   // code 2
7   nondet_assign(dbl, {1.1, 2.2, 3, 3.1});
8   // code 3
9 }

```

```

1 c_decl { char ch; double dbl; }
2
3 active proctype p() {
4   // code 1
5   byte tmp_ch_idx;
6   select(tmp_ch_idx : 0 .. 2);
7   c_code {
8     if (Pp->tmp_ch_idx == 0) { ch = 'a';}
9     ...
10    Printf("### ch = %c", ch);
11  }
12  // code 2
13  byte tmp_dbl_idx;
14  select(tmp_dbl_idx : 0 .. 3);
15  c_code {
16    if (Pp->tmp_dbl_idx == 0) { dbl=1.1;}
17    ...
18    Printf("### dbl = %lf", dbl);
19  }
20  // code 3
21 }

```

Fig. 6 The `nondet_assign` concept and its encoding. The example uses discrete elements of type `char` and `double`. In the translated Promela code, the first step is to choose the index of the element to be assigned (line 6) and then the assignment is performed accordingly (line 9). Additionally, lines 15 and 30 contain automatically inserted `Printf()` function calls in the special witnesses format

named `var` with values between 200 and 300, using the random generator seed 2. Line 5 encodes the initialisation of the random generator with the chosen seed; lines 6–9 implement the generation of 50 random numbers; finally, lines 10–13 implement the assignment of the variable with the random value. Again, the random assignment concept is able to deal with variables of different types.

5.1.5 Assumption concept

Frequently, the requirements on 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 subsequent statements to be 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. A harness can have multiple

```

1 decls { long var; }
2
3 harness p() {
4   // code 1
5   random_assign(var, [200, 300], 2) : 50;
6   // code 2
7 }

```

```

1 c_decl { long var; }
2
3 active proctype p() {
4   // code 1
5   c_code { srand(2); }
6   do
7     :: cnt < 50 -> cnt = cnt + 1;
8     :: break;
9   od;
10  c_code {
11    var = (rand() % (300 - 200)) + 200;
12    Printf("### var = %d", var);
13  }
14  // code 2
15 }

```

Fig. 7 The `random_assign` concept and its encoding. The Promela translation makes use of the `srand()` and `rand()` C code functions to generate random values (lines 10 and 18)

```

1 harness p()
2 {
3   // code 1
4   assume(cond);
5   // code 2
6 }

```

```

1 active proctype p() {
2   // code 1
3   end_label: (cond);
4   // code 2
5 }

```

Fig. 8 The `assume` concept and its encoding via an “end-state” label. If the assumed condition does not hold we force a backtracking step

assumes interleaved with other code. Figure 8 illustrates the concept and its translation to Promela. The encoding makes use of the `end-state` label of Promela which marks a local control state that is acceptable as a valid termination point—if the condition evaluates `false`, then the execution is terminated and a back-step enforced.

5.1.6 Non-deterministic choice concept

It is frequently necessary to select the next behaviour of the harness logic for execution in a non-deterministic manner. We support this need via the `nondet_choice` concept which provides the possibility to specify multiple guarded alternative behaviours. A choice succeeds if the guard of the selected behaviour is true. When none of the guards evaluate to true, a default no-op branch is executed, see Fig. 9.


```

1 harness p()
2 {
3   // code 1
4   nondet_choice {
5     choice: guard1 -> { /* code 2 */ }
6     choice: guard2 -> { /* code 3 */ }
7   }
8   // code 4
9 }

```

```

1 active proctype p() {
2   // code 1
3   if
4     :: guard1 -> {
5       c_code { Printf("### Choice: 1"); }
6       /* code 2 */
7     }
8     :: guard2 -> {
9       c_code { Printf("### Choice: 2"); }
10      /* code 3 */
11    }
12    :: else -> {
13      c_code{Printf("### Default choice");}
14    }
15  fi;
16  // code 4
17 }

```

Fig. 9 The `nondet_choice` concept and its encoding as a Promela `if` statement. The `nondet_choice` statement has a mandatory default branch which is encoded as an `else`

The encoding in Promela is done via an `if`-selection construct with an additional `Printf` for logging the path in the witness. From `nondet_choice`, we generate an `else` construct to ensure that the execution does not block if all guards evaluate to false.

5.1.7 Further concepts

Besides the concepts listed above, the 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 definition language into Promela encounters references to C code, it wraps them automatically with proper `c_code` or `c_expr` constructs.

5.2 Advanced harness definition language concepts

The basic harness definition concepts presented in the previous section can be used to define advanced harness definition constructs. This approach is demonstrated for the combinatorial testing extension. Another example of an advanced concept is the approach to support systems with hidden internal state. Because of its practical importance and speciality of its solution, it is discussed separately in the next section.

```

1 harness p() {
2   combinatorial (M) {
3     par_1 : {V_11, ..., V_1P_1}
4     ...
5     par_N : {V_N1, ..., V_NP_N}
6   }
7
8   execute_suv(par_1, ..., par_N);
9 }

```

```

1 harness p() {
2   nondet_assign(choice_par_1, {0, 1});
3   ...
4   nondet_assign(choice_par_N, {0, 1});
5
6   assume(choice_par_1 + ...
7         + choice_par_N == M);
8   if
9     :: choice_par_1 == 1 ->
10      nondet_assign(par_1,
11                    {V_11, ..., V_1P_1});
12    :: else -> par_1 = V_11;
13  fi;
14  ...
15  if
16    :: choice_par_N == 1 ->
17     nondet_assign(par_N,
18                   {V_N1, ..., V_NP_N});
19    :: else -> par_N = V_N1;
20  fi;
21
22  execute_suv(par_1, ..., par_N);
23 }

```

Fig. 10 The `combinatorial` concept and its encoding. First we choose all combinations of m parameters (lines 2–7) and then for a chosen combination we non-deterministically assign corresponding values (lines 8–20)

5.2.1 Combinatorial testing

A highly popular test method is combinatorial testing [17]. For a system which takes a large number of n parameters, each having discrete values, m -wise combinatorial testing, $m \leq n$, triggers the system with all combinations of any m parameters. If m is 2 then we have “pairwise testing”, if m equals with n then we explore exhaustively the entire input space. Choosing an appropriate value for m is an engineering question which trades the input space coverage against execution time.

In Fig. 10, we present the high-level concept (upper part) and its encoding in terms of the lower level language concepts (lower part of the figure). In the encoding, we firstly choose all possible sets of m parameters (lines 2–7). Next, all possible combinations of values for the parameters contained in the current set are non-deterministically generated (lines 8–20). Extensions of combinatorial testing like the mixed strength generation [39] or the simple exclusion of forbidden combinations can be implemented by adding additional

assume statements to the harness definition before SUV execution, see Sect. 7.2 for an example.

Note that the proposed implementation of combinatorial testing does not necessarily generate the smallest subset of m -wise combinations, which is an optimisation problem [17]. Though it demonstrates the expressiveness of non-deterministic specifications and the power of the Spin model checker. Moreover, in lines 2–4, we generate all combinations of parameters (2^N) and only afterwards filter them using the assume construct in line 6. If N is big, then this generation can be expensive in terms of execution time. In this case, the generated code should be optimised with additional assumptions to prevent the choice of more than m members.

5.3 Verifying systems with internal state

Spin's variables tracking capability to restore the previous state of the system under verification when backtracking is essential. However, it implies that the internal state of the SUV is accessible and can thereby be managed by Spin. This is not always the case since many systems encapsulate internal state and make it non-accessible and even non-observable from the exterior, e.g. in case of hidden global variables, use of third-party libraries or internal dynamic memory allocation. In the following, we deal with the two cases of accessible and hidden internal SUV state.

5.3.1 Accessible internal state

When the SUV exhibits internal state that is represented in terms of global variables accessible by the user through the source code, these variables need to be controlled. For this purpose, the `track state` statement can be used, see Sect. 5.1. Figure 11 illustrates this approach. It assumes that the SUV state variable called `crtState` is tracked during execution. If a SUV call changes the internal state, the tracked state variable changes its value. Because its value change is observed by Spin, it can be restored when backtracking occurs and an error trail, which is generated in case of a failure, becomes repeatable. In addition, we introduce the `multistep` concept into the harness definition language. It assumes that the statement body calling the SUV is repeatedly executed N times. The `multistep` statement is translated in Promela to a loop. The iteration number is logged to ease understanding the witness.

5.3.2 Hidden internal state

Tracking a SUV state that is unobservable in the verification harness, such in case when using third-party libraries that are available only as binary code, requires a different approach. First, we assume that the SUV exhibits a *reset* function that drives the SUV back to its known initial state from any state

```

1 track state: crtState;
2 harness p() {
3   // init
4   multistep(N) {
5     // body
6   }
7 }

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

```

Fig. 11 The `multistep` concept and its encoding. The body of the statement is copied N times in Promela; also witness information about the current iteration is printed (line 9). The `track state` statement illustrates the tracking of an accessible SUV state variable

it is currently in. That is, we consider only this kind of resettable software systems. The assumption of a reset function is common in software testing and can be frequently met in practice, e.g. by re-initialising the computer memory through a new instance of the SUV.

Figure 12 illustrates the verification harness for this case. Similar to the case when state variables are accessible, the harness makes use of the `multistep` statement. Additionally, the harness is attributed with a reference to the SUV reset function, called `suv_init` (line 1). The body of the `multistep` contains the SUV calls (e.g. `suv_fun1` and `suv_fun2`) that might cause hidden SUV state transitions.

The translation to Promela replaces the calls to the SUV functions specified in the harness with proxy calls, denoted as `*_proxy()` (lines 11–12), that are implemented in a generated wrapper. Furthermore, the code contains additional track statements for counter variables (lines 1–4) also defined in the wrapper.

The *wrapper* is a C file generated from the harness definition; see the sketched code in Fig. 13 for the example from Fig. 12. Its purpose is to ensure consistency and repeatability of the verification run. It achieves these properties by storing the call sequence of all SUV calls in the global variable `calls_history` (line 7) and tracking the current call sequence depth (line 2). Similarly, if a SUV function takes parameters, such as function `suv_fun2` in Fig. 12, the actual parameter history and the current number of calls of this function are also stored (lines 10 and 3).

```

1 << reset function : suv_init >>
2 harness p() {
3   // code 1
4   multistep(N) {
5     // code 2
6     nondet_choice {
7       choice: { suv_fun1(); }
8       choice: { suv_fun2(arg); }
9       ...
10    }
11  }
12 }

1 c_track "&suv_calls_cnt"
2   "sizeof(uint16_t )" "UnMatched"
3 c_track "&suv_fun2_calls_cnt"
4   "sizeof(uint16_t )" "UnMatched"
5
6 harness p() {
7   // code 1
8   multistep(N) {
9     // code 2
10    nondet_choice {
11      choice: { suv_fun1_proxy(); }
12      choice: { suv_fun2_proxy(arg); }
13      ...
14    }
15  }
16 }

```

Fig. 12 Harness definition for a system with hidden internal state. The user must specify the SUV reset function for re-initialisation. The translation of the harness into Promela is performed in two steps. First, calls to SUV functions are replaced by proxy calls (lines 11–12) and some variables of a special wrapper are tracked (lines 1–4). In the next step then, multistep statement is translated

The approach works in combination with the state-space traversal strategy implemented in Spin. When Spin arrives in a state, in which a SUV function shall be called, its corresponding proxy function is called instead. The proxy function saves the current arguments and replays the history of calls as shown in `do_perform_call`. This function resets the SUV to its initial state by calling `suv_init` (line 25). From here the proxy function repeats all function calls stored in the history list `calls_history` (line 7) in the given order up to the depth maintained in `suv_calls_cnt` (line 2). When a function has arguments, calls to this function use the arguments saved in the history list `arg_hist`, line 38. Assuming that the SUV does not contain any internal non-determinism, the SUV will faithfully perform the same state transitions as it did before.

When Spin backtracks to a state with unexplored transitions, it will modify the `suv_calls_cnt` variable which tracks the total number of calls and the `suv_fun2_calls_cnt` variable which tracks the number of calls for each function receiving arguments. The variables will receive the same value which they had when Spin arrived at this state at the first time. Given that backtracking happens only to states visited

```

1 // how many times a function was called
2 uint suv_calls_cnt;
3 uint suv_fun2_calls_cnt;
4
5 // history of function calls
6 enum { SUV_FUN1, SUV_FUN2 }
7   calls_history[MAX_HIST_SIZE];
8
9 // history of arguments
10 int16_t arg_hist[MAX_HIST_SIZE];
11
12 void suv_fun1_proxy() {
13   calls_history[suv_calls_cnt++] = SUV_FUN1;
14   do_perform_call();
15 }
16
17 void suv_fun2_proxy(int16_t arg) {
18   arg_hist[suv_fun2_calls_cnt++] = arg;
19   calls_history[suv_calls_cnt++] = SUV_FUN2;
20   do_perform_call();
21 }
22
23 void do_perform_call() {
24   // (re-)initialize the SUV
25   suv_init();
26
27   uint crt_call_to_fun2 = 0;
28
29   // re-play all calls saved so far
30   for (uint i=0; i < suv_calls_cnt; i++) {
31     switch(calls_history[i]) {
32       case SUV_FUN1: {
33         suv_fun1();
34         break;
35       }
36       case SUV_FUN2: {
37         suv_fun2(
38           arg_hist[crt_call_to_fun2++]);
39         break;
40       }
41     }
42   }
43 }

```

Fig. 13 Wrapper generated around the used SUV API. For each SUV function, we generate a proxy function that memorises the calls to this function, including the values of its actual parameters. Global variable `calls_history` (line 7) is used to save the call sequence. For each argument of a SUV function we generate a global variable `arg_hist` (line 10) that saves the history of actual values used in calls

before, the value of one of the counter variables is smaller than it was before. Hence, the historical call sequence will be shorten and a new value for the last SUV function call will be added. This last call is likely different from the old one stored at this position (because Spin decided to explore another branch).

On the one hand, the penalty of this approach is an overhead of resetting the SUV and repeated re-executions of previously called SUV functions. On the other hand, it is completely transparent to Spin's internal state exploration strategy which does not need to be modified. Moreover, the approach is general enough to deal with any SUV components, in particular components that contain third-party libraries of unknown behaviour. The only assumption for the

correct functioning of the harness is the availability of the SUV reset function.

6 Combining MDCC with code sanitizers

An important category of software bugs are related to runtime errors and undefined behaviours causing programs to crash or being exploitable in security attacks. Modern compilers offer the possibility to instrument programs with sanity checks, called *code sanitizers*, that are subsequently performed when the code is executed at runtime. This section discusses the integration of the MDCC approach with code sanitizers.

6.1 Clang sanitizers

Sanitizers check for a broad range of properties like the presence of undefined behaviour in the code or for defects like those related to memory. The technology is well supported in the LLVM/Clang compiler project for C and C++. Similar to any runtime verification approach, the success of using code sanitizers to identify errors in code highly depends on the input data used to exercise the code. Furthermore, when runtime checks fail, developers need diagnosis data about both the failure location and the inputs which caused that failure.

Deploying a sanitizer in an MDCC approach requires therefore only to generate inputs and invoke calls to the SUV since the verification conditions are instrumented by the sanitizer itself. When the sanitizer fails, Spin needs to be informed to trigger the generation of the error trail.

A broadly used sanitizer is the *Undefined Behaviour Sanitizer* (UBSan).⁵ UBSan can identify the occurrence of undefined behaviours such as overflows or the use of null pointers. The recent version of UBSan shipped with the Clang compiler supports more than 20 different checks. When a sanity check fails during execution of the instrumented code, a *trap* function is activated. The default implementation of the trap prints a diagnosis message on the console and then exits the program. In addition UBSan offers the possibility to specify a custom trap function and thereby control the reaction when a sanity check is violated. In the following section, we demonstrate the integration of a sanitizer using the UBSan as example.

6.2 Integrating sanitizers with Spin

A custom trap function, provided by *mbeddr-spin*, calls the `uerror()` internal function of Spin in order to trigger the generation of an error trail. In Fig. 14, we illustrate the command line used to compile the SUV and the *pan* program

```
1 clang -g -fsanitize=undefined
2       -fsanitize-trap=undefined
3       -ftrap-function=spin_sanitizer_trap
4       pan.c custom_trap.c suv.c
```

```
1 // custom_trap.c
2 void spin_sanitizer_trap() {
3     uerror ('sanitizer error');
4 }
```

Fig. 14 The command line used to compile the SUV, *pan* and custom trap code using `clang` (top). The custom trap function calls the internal `uerror()` function of Spin with an error message that will reappear in the error trail (bottom)

of the verification harness with the sanitizer and the custom trap function. The corresponding options are:

- `-fsanitize=undefined` selects which sanitizers are used (the name `undefined` refers to UBSan),
- `-fsanitize-trap=undefined` instructs the sanitizer to call a trap in case of a failed check, and
- `-ftrap-function=custom_trap_func` instructs the sanitizer to call `custom_trap_function`.

No further efforts are required to make the integration of sanitizers into MDCC work. Triggering the error trail generation for the counterexample is sufficient. The error trail is then interpreted and lifted to the abstraction level of the verification harness definition language in the same manner as discussed in Sect. 4.3. The approach outlined above can be easily adopted to integrate other code sanitizers of Clang, e.g. Address Sanitizer or Memory Sanitizer. As of version 5.0 of Clang, which we integrated with Spin, these other sanitizers did not directly support the definition of custom *trap functions* and thereby replaying of counterexamples will not work.

7 Examples of covered software components



In this section, we present examples of categories of software components that are covered by our approach. Each subsection below deals with one category and demonstrates how the verification harness can be modelled using our language. The code examples themselves are taken from real-world projects and are typical for a broad range of software code encountered in practice.

7.1 Cat I: functions with in/out parameters

This category describes cases when the SUV interface is represented by a side-effects free function, which possesses some parameters that are used both as inputs and outputs.

⁵ <https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html>.

```

 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)

```

Fig. 15 Example of a harness definition for verifying a function with in/out parameters. All in/out parameters must be tracked to restore their input values when backtracking

Here, the harness must track the variable which is used as an actual in/out parameter in the call of the function.

Figure 15 presents an example of such a harness that verifies a sorting algorithm `heap_sort`. Sorting of elements provided in an array occurs in place. First, an array with five elements is non-deterministically initialised with values between -10 and 10 ; then the sorting function is called; and the returned array is checked for correctness. The state of the array that serves as in/out parameter must be tracked through `track state` such that, when backtracking is performed during execution, the search algorithm can continue from the last state stored in this array before it was called. In this example, backtracking happens each time after an assignment to the array was processed—the sorting function was called and the verification condition checked. After backtracking, a new assignment of array elements is produced and the SUV is called again.

7.2 Cat II: combinatorial testing

When software systems have a large set of input parameters, then combinatorial testing can be a viable trade-off between input coverage and efforts spent. A prerequisite to this approach is that discrete values for each parameter have been selected which are used in combination with other parameter values as inputs to the SUV. Additional `assume` statements can be used to ensure the validity of parameters combinations.

Figure 16 presents the harness for verifying the conversion functions `mktime` and `localtime` from the C library. The `mktime` function takes as argument a structure with many fields representing date and time information and transforms it to an integer representation. The `localtime` function transforms the integer value back to the data and

```

harness time_harness {
  combinatorial (3) {
    my_mins : { 0, 10, 30, 59 };
    my_hour : { 0, 1, 6, 12, 18, 23 };
    my_days : { 1, 10, 20, 28, 29, 30, 31 };
    my_month : { JAN, FEB, APR, MAY, DEC };
    my_year : { 71, 100, 110, 111 };
  }

  assume(my_month == FEB -> (my_days < 30));
  assume(my_year % 4 != 0 && my_month == FEB -> (my_days < 29));
  assume(my_month == APR -> (my_days < 31));

  my_tm.tm_min = my_mins;
  my_tm.tm_hour = my_hour;
  my_tm.tm_mday = my_days;
  my_tm.tm_mon = ((uint8) my_month);
  my_tm.tm_year = my_year;

  my_time = mktime(&my_tm);
  my_tm_back = localtime(&my_time);

  assert(my_tm_back->tm_min == my_tm.tm_min);
  assert(my_tm_back->tm_hour == my_tm.tm_hour);
  assert(my_tm_back->tm_mday == my_tm.tm_mday);
  assert(my_tm_back->tm_mon == my_tm.tm_mon);
  assert(my_tm_back->tm_year == my_tm.tm_year);
} time_harness (function)

```

Fig. 16 Example of a harness definition for verifying time conversion functions. We use combinatorial testing with additional constraints expressed in `assume` statements

time structure. Besides the `combinatorial` construct which handles the computation of all 3-wise combinations of parameter values, `assume` statements constrain the relation between values for months and days.

7.3 Cat III: systems with accessible internal state

Another category encountered in practice is represented by cases when the software component contains internal state and its behaviour depends on this state. Here, we consider the sub-case of systems whose state, typically expressed through global variables, is accessible within the verification harness by referencing the global variables directly. An example for this category are implementations of state machines. Their implementations frequently make use of a stepper function that takes the input event as a parameter and triggers the execution of the next state transition accordingly.

Figure 17 depicts an example of a harness for state machine verification. The SUV contains the `init_sm()` function to initialise the state machine and the `do_step()` function for stepping through. Because the harness needs to reason about the correct state after each step, all variables related to the SUV state must be tracked in the harness—e.g. the `crtState` variable declared in the SUV. The `multistep` statement ensures the continued execution of the state machine for the given number of steps. In order to facilitate the understanding of a counterexample, the `crtState` variable denoting the current state is logged in the witness.


```

statemachine_harness constraints
imports ① 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)

```

Fig. 17 Example of a harness definition for verifying a state machine deploying track state and multistep statements

```

decls {
  #constant ELEMENTS_IN_TREE = 10;
  int8[ELEMENTS_IN_TREE] elements;
  uint32 my_count;
  int8* last_added_element;
}

track state: &my_count;

<< generate wrappers for SUV E tree234 >>
<< reset and replay, reset function :re_init_tree >>
harness tree234_harness {
  my_root = newtree234_root(:cmp_uint8);
  nondet_assign(elements, { -100, -1, 0, 1, 100 });

  for (uint8 idx = 0; idx < ELEMENTS_IN_TREE; idx++) {
    // "add234(e) - should return e on success, or if an existing "
    // element compares equal, returns that.
    last_added_element = ((int8*) add234(my_root, &elements[idx]));
    assert(*last_added_element == elements[idx]);

    // check if the tree contains already an element with same value
    if (last_added_element == &elements[idx]) {
      my_count++;
    } if
    assert(my_count == count234(my_root));

    for (uint8 idx2 = 0; idx2 < idx; idx2++) {
      assert(find234(my_root, &elements[idx2], :cmp_uint8) != NULL);
    } for
  } for
} tree234_harness (function)

```

Fig. 18 Example of a harness definition for verifying a 2-3-4 tree. Because the SUV allocates memory, a wrapper must be generated which resets the SUV using the function `re_init_tree` and replays the execution sequence recorded so far each time a SUV function is called

7.4 Cat IV: systems with hidden internal state

The other sub-case of state-based systems deals with systems whose state is hidden to the verification harness. Examples comprise systems whose source code is not available or use additional libraries that contain state. Also components allocating memory dynamically belong to this category.

Figure 18 presents a harness for verifying properties of a 2-3-4 tree implementation. This component offers an API for adding elements to the tree, searching for existing elements, deleting elements and counting the number of stored elements. When a new element is added to the tree, the implementation allocates memory dynamically, which cannot be tracked by the harness. The high-level harness definition contains two annotations `<<generate wrappers...>>` and `<<reset function...>>` which instruct the generator to generate code to reset the SUV using the function `re_init_tree` and replay the inputs. The functionality of input replay is explained in Fig. 13.

The harness non-deterministically assigns values to an array of elements. Then, in a `foreach` loop, each element from the array is added to the tree. The subsequent assertions cover three different aspects: (1) the function `add234` shall return either the element added or a pointer to an element in the tree with the same value if it already exists; (2) the function `count234` shall return the correct number of elements contained in the tree; and (3) the function `find234` shall find all elements in the tree that have been added so far.

7.5 Cat V: composition of state machines

Many systems (e.g. communication protocols, producer/consumer systems) possess behaviour which is implemented as communicating asynchronous state machines. Consider the following example in Fig. 19. It depicts the architecture of a fault-tolerant system realised through two micro-controllers equipped with sensors and a communication link between them (dual-channel architecture). Each controller monitors a health level of its periphery. It is assumed that at any given moment in time only one controller is allowed to control the actuators. This controller shall be the one with the best health level as calculated from its sensors. The selection of the “master” controller is managed through a distributed `master_selection` algorithm. Each controller runs its own instance of the algorithm implemented as a state machine. We are interested in verifying this selection algorithm, which is a special solution to the classical leader election problem [10].

The middle part of Fig. 19 presents the data structure holding the internal state of each controller: `status` models the current status of the controller (it can be master, slave or in intermediate state from master becoming slave), `health_level` is the health level of the current controller, `master_req` informs that the current controller requests the master role, `master_req_ack` is true if the

current controller acknowledges a master request from the other controller. The state machine implemented by the `master_selection` function runs on each controller and has read/write access on this data structure while having only read access to that data, i.e. this is the state of the current controller while that is the state of the other controller.

The lower part of Fig. 19 presents the harness. At the beginning, `comp1` holds the master role and both controllers are in best health (health level is 3). We then perform a multistep verification and simulate asynchronous executions of the two instances of the algorithm via the `nondet_choice` concept. At each step, the health status of each controller is assigned non-deterministically a different value ranging from 1 (bad) to 3 (very good). Finally, the assertions check that only one controller is in MASTER, SLAVE or INTERMEDIATE state respectively. Finding the right depth of the multistep requires knowledge about the code structure of the algorithm. An enhanced version of this verification harness could add fault-injection to further verify robustness of the algorithm or, depending on the needs, define degradation models for sensors, e.g. assuming that the sensor health can only decrease but never increase.

7.6 Cat VI: systems with huge input space

We often encounter situations when the expected verification outcome expressed as an oracle has a complex structure and cannot be captured through simple assertions or temporal logic expressions. An example for this category are algorithms that compute Cyclic Redundancy Check (CRC) values. 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 data and the occurrence of error bursts. Coping with this challenge, we combine random testing with exhaustive verification. A compromise has to be found to treat the input space of some parameters exhaustively, while performing only a random selection of values for the others.

In Fig. 20 (left), we present a harness for checking the implementation of a CRC16 algorithm. The payload of a message is initialised 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 CRC16 value. Afterwards, all possible burst errors up to length 13 are applied on the message at any location inside the payload. The start of the burst error is given by a `nondet_assign` construct. In the next `foreach` loop, the possible error bursts are simulated. At a certain bit position, an error might be introduced or not—encoded via a `nondet_choice` statement. The CRC16 value for the mutated payload is computed again.

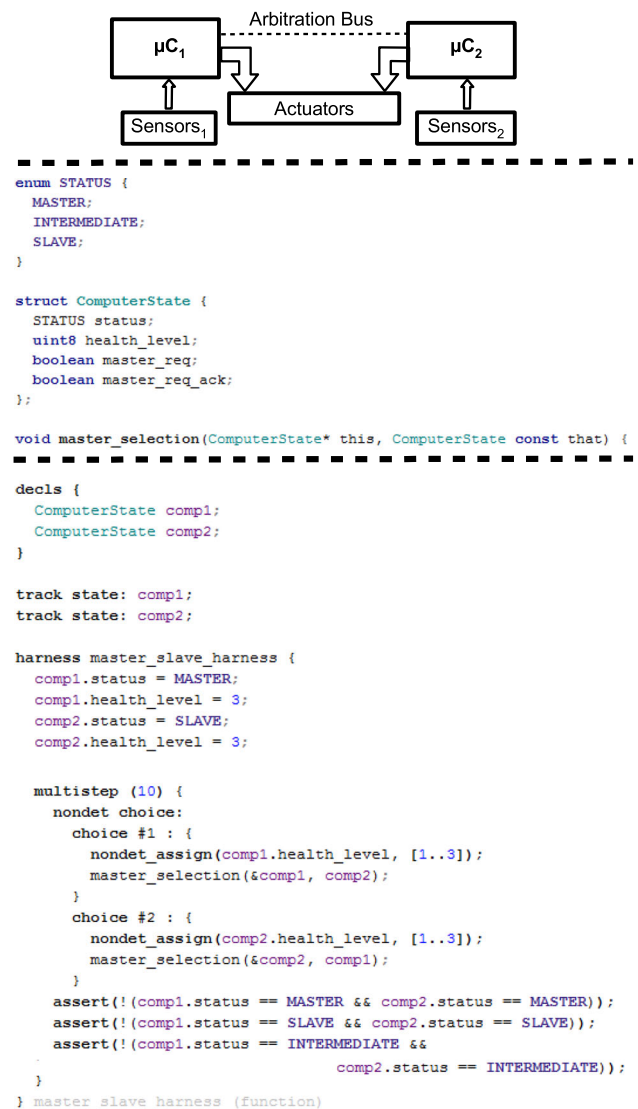


Fig. 19 Example of a fault-tolerant system using a dual-channel master-slave architecture (up). Two state-machines running in parallel on each controller select the current master based on the health status of the controllers (middle). The harness for verifying the `master_selection` algorithm (down)

It is expected that the new value differs from the original one under all considered error bursts, which is formulated in the corresponding `assert` statement. The log witness statements serve as additional information in a generated counterexample.

Figure 20 (right) presents a witness returned by Spin and lifted to the harness language. Due to the introduced format of the logging construct, the tool `mbeddr-spin` can establish the relation between a witness entry and the corresponding statement in the verification harness. By double-clicking on an entry, the corresponding harness part, from which that witness entry originates, is selected automatically. This automatism makes it easier to understand big witnesses and navigate through the harness code.

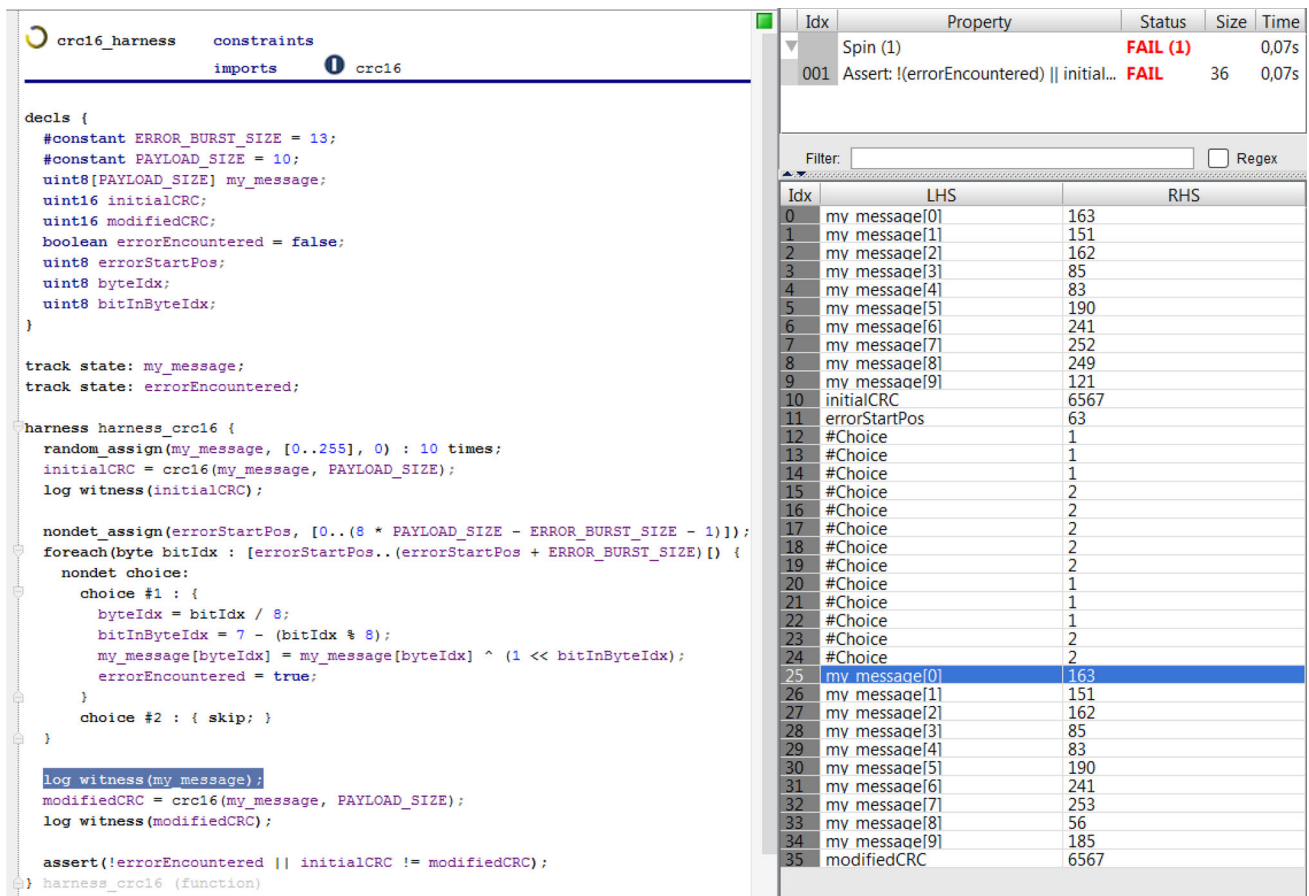


Fig. 20 Harness definition for checking a CRC16 algorithm (left) and the presentation of a witness (right). A double-click on a witness entry selects the corresponding node in the harness definition. Additional logging of harness variables occurs in the witness and facilitates its understanding

8 An industrial example

In this section, we present our experiences from the application of our approach for the verification of a safety-critical software component of the European Train Control System (ETCS) at Siemens, which is a signalling and control component of the unified rail traffic management system in Europe.

8.1 Problem domain

One of the central safety requirements of ETCS deals with the probability of the unwanted event that a train overruns a so-called “danger point”, e.g., by entering a block that is already occupied by another train. To avoid this case, the train must come to a halt before that point. Here, the safety of the system relies on the Emergency Brake Deceleration curve that must be implemented in trains with different confidence levels. A key characteristic of the braking curve is the safety factor K_{dry} that has been introduced with Baseline 3 of the ETCS specification [20]. It determines the maximum braking delay when braking on dry rails. The K_{dry} factor is dependent on the characteristics of the braking system of a certain

train configuration and the confidence level of the emergency brake safe deceleration (EBCL). For each speed level and for each EBCL, the train manufacturers must specify a certain value for K_{dry} which represents the guaranteed minimum deceleration coefficient. The values of this factor, depending on many train parameters, are of capital importance for the safe functioning of the braking system.

The computation of the K_{dry} factor is performed by a software component which implements a Monte-Carlo simulation as described in [26,32]. The variabilities of the different physical parameters of the train are described via statistical distributions. The selection of the parameters and their distributions are based on experimental data and on the knowledge of braking experts. All in all, about 30 parameters are used as input for the simulation. To get enough confidence, the computation of K_{dry} for each set of train parameters is performed by running at least 10^9 independent simulations.

To ensure that the K_{dry} computation is performed correctly, the algorithm was implemented by two independent engineers working for two different units at Siemens. Both of these implementations use the same input parameters, but the structuring of the code and the used libraries for random

numbers generation are however different. One implementation uses the PRNG library for pseudo-random numbers generation⁶ and the other one the algorithms for generating random numbers provided by the GSL library.⁷ The core part of both simulations contains approximately 500 lines of C code using complex computations involving deeply nested loops, values of double type and function calls to the random numbers generation libraries. Checking the equivalence of these algorithms using static analyses tools is infeasible and the case for using a formal approach by deploying Spin-based model-driven code checking was quickly taken up (see Sect. 3.1).

8.2 Verification problem

The primary task was to check the equivalence of both algorithms on a wide variety of train configurations considered relevant by the domain experts. A secondary goal was to make sure that the implementations do not exhibit runtime errors or any undefined behaviour for the given configurations. We performed these two tasks using the approach described in this paper.

8.2.1 Defining the harness

For defining the harness, we used the following DSL constructs: non-deterministic assignments for creating possible train configurations, assumptions for ruling out invalid configurations and logging constructs to make the witness easier to understand. The harness was built iteratively as a joint work with domain experts—initial versions were too permissive which resulted in code faults that were dismissed by domain experts because they represented impossible combinations of values of configuration parameters. The final version of the harness covers more than 10,000 valid train configurations. In the DSL, it is only 25 lines long, while the generated Promela code has 160 lines. Defining the harness proved to be easy; debugging the reasons for failures was however a labour-intensive task. Furthermore, additional efforts were required to unify the interfaces of the two simulation programs used as SUVs to make them callable from the same harness.

8.2.2 MDCC with code sanitizers

We used the Clang sanitizers to check the absence of runtime errors and undefined behaviour as outlined in Sect. 6. There had been several cases of configurations, for which the implementations exhibited undefined behaviours. Further

investigations together with the domain experts revealed that these cases refer to locations in the code that were not covered with the previously used test set of configurations and by (manual) code reviews. Only the exhaustive exploration of all of these configurations made it possible to identify these cases of undefined behaviour. Saving the witness as an error trail eased considerably the work to reproduce an observed case and discuss it with the domain experts.

8.2.3 Equivalence checks

The check for equivalence was performed by comparing the outcome from both simulation algorithms for all given configurations. However, performing 10^9 independent simulations for each of the roughly 10,000 possible configurations would be prohibitively expensive. We thereby decided to restrict the simulation number to 10^3 runs and checked the equivalence of the algorithms after this number of simulations. We could identify only small differences between the results of these algorithms, in all of these cases the differences being explainable by the statistical dispersion due to the relatively small number of simulations.

9 Discussion

In the following, we provide a short discussion based on preliminary experiences with our approach.

9.1 On practical applicability of MDCC

A major limiting factor in software validation is the available time for this activity. Analysing the runtime performance of the generated `pan` executable which represents the verification harness with the integrated SUV, one observes that it is mostly constraint by the execution time of the called SUV functions. However, performing myriad calls can quickly exhaust the available time resource. The total number of SUV function calls therefore needs to be controlled by the user.

Consider the verification of the CRC16 algorithm (Fig. 20). The verification harness applies 10 different payloads, each 10 bytes long. The generation of mutated payloads with error bursts up to length 13 produces $(8 \cdot 10 - 13) \cdot 2^{13} = 548,864$ modified payloads. We experimented with Spin 6.4.9 which was able to cover this amount of data in a few seconds.

Our experiments show further that the overhead needed by the proposed implementation of the combinatorial construct is rather small. For example, when the total number of parameters n is 20 and the number of parameters considered for combinations m is 3, each parameter having 5 discrete values, the harness generated the combinations (142,500 in total) in ca. 5 s. Bigger values of n or m would however soon prove to be prohibitively expensive.

⁶ <http://statmath.wu.ac.at/software/prng/>.

⁷ <https://www.gnu.org/software/gsl/>.

When considering systems with inaccessible internal state, the reset-and-replay approach, outlined in Sect. 5.3, proves to be very expensive. For example, checking the `tree234` implementation from Fig. 18 took about 430 s—the generation of the 5^{10} inputs alone via `nondet_assign` took about 7 s. This is an intrinsic problem with the chosen strategy and might restrict its applicability in practice.

On the other hand, our approach allows a user to experiment with different strategies to find the right balance between execution time and coverage. For example, he could exchange the exhaustive input data generation of the `nondet_assign` concept with randomly generated data using the `random_assign` concept. This flexibility is reached essentially through the chosen language engineering technologies.

9.2 On usability of the tool

We conducted this work in order to enable developers benefit from the application of formal methods by hiding most of the involved complexity under the hood of higher-level language abstractions and tooling.

Using the harness definition language allows the user to write compact harness code which is then expanded into verbose and complex Promela models with intertwined C code. In Table 1, we present the lines of code of the harness written in our DSL and the generated Promela model. In the generated code, we removed manually the empty lines and performed small post-processing where the generator was too verbose. We remark that the generated Promela code is 3–8 times longer. This means that despite the fact that similar Promela code, if written manually, could be more condensed, applying the DSL results into more compact harness description.

Writing efficient verification harnesses requires still expertise in the underlying principles of non-deterministic modelling and verification, though. A way out of this problem is that a verification expert interacts constantly with domain experts when creating the harness. This approach turned out to be successful when we were working on the K_{dry} example. Conversely, once the harness definition was created, the domain experts were able to understand and validate the harness without major difficulties. Furthermore, when a verification run failed, the generated high-level witness can be used to discuss the failure directly on statements of the harness definition. The verification expert can then transform the witness into a test vector and use it for debugging.

9.3 On relation to Spin

In this paper, we show that a relatively small set of high-level language concepts implemented as a proper DSL is enough to describe a wide range of possible harness definitions. The

Table 1 Comparison of the size of the harness written using the DSL and the size of the generated Promela code

Example	DSL size	Promela size
Heapsort—Fig. 15	11 lines	81 lines
Time—Fig. 16	25 lines	203 lines
Statemachine—Fig. 17	20 lines	165 lines
Master/Slave—Fig. 19	24 lines	90 lines
CRC16—Fig. 20	32 lines	130 lines
K_{dry} —Sect. 8.2	25 lines	160 lines

use of Spin as the underlying model checker proves beneficial due to its possibility to intertwine Promela model code with low-level C code and the general high maturity of the tooling. We use however only a relatively small subset of Promela and Spin features. Moreover the approach is not bound to Spin but it could be implemented using other state exploration engines as well. This need could arise when, for example, code written in other languages than C shall be verified.

10 Related work

As discussed in Sect. 2, our approach builds upon and extends the work presented in [11, 15, 16]. In the following, we present additional related works and their relation to our approach.

10.1 Languages for MDCC

The paper [33] presents an approach that uses a high-level DSL and Spin to check C code. The verification harness is automatically generated from a temporal specification in this DSL. After the verification run, result and witness are lifted at the level of the DSL. The paper [38] uses UML class diagrams and state-charts to describe environment models. These descriptions are subsequently translated to Promela.

Similarly, the work in [12, 13] 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. Descriptions in this DSL 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 previous work, our focus is on leveraging Promela and Spin's capabilities for C code verification by defining a higher-level language that captures commonly encountered patterns of harness definitions. Our language enables the simultaneous use of features for exhaustive exploration (model checking) and exemplified exploration (testing) of the verification harness. The user has full flexibility to deploy the language feature that fits best to a given problem.

Another body of work deals with Promela language extensions. Paper [29] presents several light-weighted patterns for using Promela. The patterns are candidates to be lifted to higher-level language concepts. The work in [8] presents an extension of Promela with stronger type checks, while the work in [19] applies language engineering technologies to implement new language features that can be seamlessly added to Promela. Our tool *mbeddr-spin* uses the same approach to leverage model-driven code checking.

10.2 Language engineering and IDEs

There has been already work done to integrate Promela into common IDEs. For example, papers [5] and [37] present Eclipse-based IDEs for Promela. Their solutions 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 integrations into the Eclipse IDE offer modern tool support for building Promela models, none of these contributions support the capability of embedding 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.

Compared with this approach, we deeply integrate Spin into the *mbeddr* environment, which already offers support for C code development. Therefore, our approach makes use of the same IDE for writing C code and harness definitions using the new language. For this reason, the whole approach becomes easier to understand and use by software developers in practice. Furthermore, advanced users can write complex harnesses easier.

10.3 Related testing approaches

Due to its highly practical relevance, a large number of test derivation strategies have been devised to detect faults in software. For example, search-based testing approaches [24] deploy heuristics grounded on empirical evidence to address the problem of generating tests that carry a high chance to detect faults. Given that these approaches have a limited effectiveness in finding faults in the general case, random testing [1] is still considered as a viable test derivation strategy, for example in fuzz testing to detect security breaches [9,40].

In our work, several testing approaches (e.g. random, combinatorial testing) have been implemented on top of Promela and Spin. Furthermore, parts of the environment can be exhaustively explored and thereby the boundary between testing and formal verification can be moved as desired.

The paper [4] presents an empirical study which compares testing with symbolic model checking to detect faults in C programs. Their study shows that model checking

approaches find more bugs in shorter time. While we agree that today's software model checkers are highly developed and potent tools, the reasons that motivate our work (see Sect. 3.1) remain valid. As a matter of fact, MDCC offers some advantages over static analysis tools. In practical terms, it appears likely that testing, MDCC and symbolic model checking show best results if applied in combination.

11 Conclusions and future work

This work is part of a larger endeavour to bring formal verification closer to practitioners. We present an approach and associated integrated environment which aims at improving the usability of model-driven C code verification using Spin. To this end, we describe a verification harness definition language that is built on top of Promela and prototypically implemented in *mbeddr-spin*. We leverage the code instrumentation capabilities of the LLVM/Clang code sanitizers and show the applicability of the language on code-checking software categories that are often used in industrial applications. In addition, we present our experiences with a concrete example of an industrial software component. Deploying the harness definition language for code-checking, a user can experiment with the various language concepts to find a right balance of affordable coverage vs execution time using the *mbeddr-spin* which is built on top of the MPS language engineering platform.

Due to its modular design, the harness definition language can be extended further with new language concepts depending on needs from industrial projects that are planned in future. Moreover, we are working on a methodology and guidelines for creating harnesses to spread the application of this technique which is a viable alternative and enhancement to classical testing.

Acknowledgements We would like to thank Dorel Coman for contributing to the implementation of Spin language extensions, Ulrich Hipp for explaining us the technicalities behind the computation of K_{dry} and helping us with performing the verification, and Olivera Pavlovic for feedback.

References

1. Arcuri, A., Iqbal, M.Z., Briand, L.: Random testing: theoretical results and practical implications. *IEEE Trans. Softw. Eng.* **38**(2), 258–277 (2012)
2. Beyer, D.: 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). Springer (2016)
3. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Computer Aided Verification (CAV). Springer, Berlin (2011)

4. Beyer, D., Lemberger, T.: Software verification: testing vs. model checking. In: *Hardware and Software: Verification and Testing*. Springer, Berlin (2017)
5. Brezocnik, Z., Vlaovic, B., Vreze, A.: SpinRCP: the Eclipse rich client platform integrated development environment for the Spin model checker. In: *International Symposium on Model Checking of Software* (2014)
6. Campagne, F.: *The MPS Language Workbench*. CreateSpace Publishing, Scotts Valley (2014)
7. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (2004)
8. Donaldson, A.F., Gay, S.J.: ETCH: An enhanced type checking tool for Promela. In: *Workshop on Model Checking Software (SPIN)*. Springer (2005)
9. Felderer, M., Büchler, M., Johns, M., Brucker, A.D., Breu, R., Pretschner, A.: Security testing: a survey. *Adv. Comput.* **101**, 1–51 (2016)
10. Garcia-Molina, H.: Elections in a distributed computing system. *IEEE Trans. Comput.* **31**(1), 48–59 (1982)
11. Groce, A., Joshi, R.: Random testing and model checking: Building a common framework for nondeterministic exploration. In: *International Workshop on Dynamic Analysis (WODA)* (2008)
12. Groce, A., Pinto, J.: A little language for testing. In: *NASA Formal Methods (NFM)* (2015)
13. Holmes, J., Groce, A., Pinto, J., Mittal, P., Azimi, P., Kellar, K., O'Brien, J.: TSTL: the template scripting testing language. *STTT* **20**(1), 57–78 (2018)
14. Holzmann, G.: *The SPIN Model Checker: Primer and Reference Manual*, 1st edn. Addison-Wesley, Boston (2011)
15. Holzmann, G., Joshi, R., Groce, A.: Model driven code checking. *Autom. Softw. Eng.* **15**, 283–297 (2008)
16. Holzmann, G.J., Joshi, R.: Model-driven software verification. In: *Workshop on Model Checking Software (SPIN)* (2004)
17. Kuhn, D.R., Bryce, R., Duan, F., Ghandehari, L.S., Lei, Y., Kacker, R.N.: Combinatorial testing: theory and practice. *Adv. Comput.* **99**, 1–66 (2015)
18. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis and transformation. In: *International Symposium on Code Generation and Optimization* (2004)
19. Mali, Y., Wyk, E.V.: Building extensible specifications and implementations of Promela with AbleP. In: *Workshop on Model Checking Software (SPIN)* (2011)
20. European Union Agency for Railways: ERTMS/ETCS Baseline 3: System Requirements Specifications. SUBSET-026 (2014). Issue 3.4.0
21. Synopsys, Inc.: Static application security testing (coverity). <https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html> (2018). Accessed 9 Apr 2018
22. The Clang Team: Clang Compiler User's Manual (2018). <https://clang.llvm.org/docs/UsersManual.html>. Accessed 9 Apr 2018
23. The MathWorks, Inc.: Polyspace – making critical code safe and secure. <https://de.mathworks.com/products/polyspace.html> (2018). Accessed 9 Apr 2018
24. McMinn, P.: Search-based software test data generation: a survey. *Softw. Test. Verif. Reliab.* **14**, 105–156 (2004)
25. Molotnikov, Z., Völter, M., Ratiu, D.: Automated domain-specific C verification with mbeddr. In: *International Conference on Automatic Software Engineering (ASE)* (2014)
26. Pierre Meyer Richard Chavagnat, F.B.: Computation of the safe emergency braking deceleration for trains operated by ETCS/ERTMS using the monte carlo statistical approach. In: *Proceedings of the 9th World Congress of Railway Research (WCRR)* (2011)
27. Ratiu, D., Ulrich, A.: Increasing usability of Spin-based C code verification using a harness definition language. In: *International SPIN Symposium on Model Checking of Software, SPIN 2017* (2017)
28. Ratiu, D., Voelter, M., Kolb, B., Schätz, B.: Using language engineering to lift languages and analyses at the domain level. In: *NASA Formal Methods Symposium (NFM)* (2013)
29. Ruys, T.C.: Low-fat recipes for SPIN. In: *International Workshop on SPIN Model Checking and Software Verification*. Springer (2000)
30. Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: Address-sanitizer: A fast address sanity checker. In: *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. USENIX Association (2012)
31. Stepanov, E., Serebryany, K.: Memorysanitizer: fast detector of uninitialized memory use in c++. In: *International Symposium on Code Generation and Optimization, CGO '15*. Washington, DC, USA (2015)
32. Stephan Horn, O.P.: Chancen und möglichkeiten der monte-carlo-methode bei der bestimmung der etcs-bremskurven. In: *Eisenbahntechnische Rundschau (ETR)* (2017)
33. Sulzmann, M., Zechner, A.: Model checking dsl-generated C source code. In: *International Workshop on Model Checking Software (SPIN)* (2012)
34. Voelter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L., Visser, E., Wachsmuth, G.: DSL Engineering. <http://dslbook.org> (2013)
35. Voelter, M., Kolb, B., Szabó, T., Ratiu, D., van Deursen, A.: Lessons learned from developing mbeddr: a case study in language engineering with mps. *Softw. Syst. Model.* pp. 1–46 (2017)
36. Voelter, M., Ratiu, D., Kolb, B., Schätz, B.: mbeddr: instantiating a language workbench in the embedded software domain. *Autom. Softw. Eng.* **20**, 339–390 (2013)
37. de Vos, B., Kats, L.C.L., Pronk, C.: EpiSpin: an eclipse plug-in for promela/spin using spoofax. In: *International Workshop on Model Checking Software (SPIN)* (2011)
38. Yatake, K., Aoki, T.: Automatic generation of model checking scripts based on environment modeling. In: *International Conference on Model Checking Software (SPIN)*. Springer (2010)
39. Yu, L., Lei, Y., Kacker, R.N., Kuhn, D.R.: Acts: a combinatorial test generation tool. In: *International Conference on Software Testing, Verification and Validation* (2013)
40. Zalewski, M.: American fuzzy lop. <http://lcamtuf.coredump.cx/afl/> (2018). Accessed 29 Apr 2018

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.