IPSJ Transactions on System LSI Design Methodology Vol.11 29–45 (Aug. 2018)

[DOI: 10.2197/ipsjtsldm.11.29]

Invited Paper

Behaviour Driven Development for Hardware Design

Melanie Diepenbeck[1] Ulrich Kühne[2,a)] Mathias Soeken[3,b)]
Daniel Grosse[1,4,c)] Rolf Drechsler[1,4,d)]

Received: March 7, 2017

Abstract: Hardware verification requires a lot of effort. A recent study showed that on average, there are more verification engineers working on a project than design engineers. Hence, one of the biggest challenges in design and verification today is to find new ways to increase the productivity. For software development the agile methodology as an incremental approach has been proposed and is heavily used. *Behavior Driven Development* (BDD) as an agile technique additionally enables a direct link to natural language based testing. In this article, we show how BDD can be extended to make it viable for hardware design. In addition, we present a two-fold strategy which allows to specify textual acceptance tests and textual formal properties. Finally, this strategy is complemented by methods to generalize tests to properties, and to enhance design understanding by presenting debug and witness scenarios in natural language.

Keywords: behaviour driven development, test driven development, test generation, property generation, specification

1. Introduction

Software plays a major role in our lives – we use software in many appliances in our homes and workspaces. Since software becomes more widespread, its functional range and its complexity has increased in the last decade. As a consequence, building (new) features for customers becomes more challenging. In order to cope with this challenges new software development processes have emerged.

Behaviour Driven Development (BDD) has gained increasing attention as an agile development approach. BDD aims to bridge the communication gap between the software development team and the non-technical stakeholders. For this task both groups collaborate and create so-called *user stories* in natural language. By this, they capture the features of the software in terms of *scenarios*. Scenarios are expressed using a *Given-When-Then* sentence structure to connect the human concept of cause and effect to the software concept of input/process/output in an intuitive way [40]. By writing *glue code*, the software developer gives an executable semantics to the scenarios, which allows to run them as acceptance tests. Furthermore during the agile process developers create, rapidly test, and integrate small, valuable code sections—in case of BDD with the natural language "interface." Strong tool support has been developed in the recent years (e.g., Refs. [32], [40]).

While the hardware business is traditionally more conservative than the software world, there is an increased interest in agile design methodologies for the design of circuits and systems [24], [25], [27]. However, adapting the BDD flow to the design of integrated circuits is not a trivial task. There are several hardware-specific aspects that need to be taken into account. Some of these issues are related to the nature of *Hardware Description Languages* (HDLs), such as timing, concurrency, and the use of dedicated testbenches. A major difference in hardware development is, however, the demand for first-time-right designs since hardware re-spins are extremely expensive.

In BDD, the system under development is checked using acceptance tests, i.e., scenarios that test whether certain features are implemented according to the requirements. But, when considering safety critical hardware designs, testing is not enough. Formal methods are necessary to find subtle bugs that might be missed when the design is simulated, covering only few selected inputs. While in testing, tests are used to describe part of a requirement, in formal verification *properties* are the key elements that formalise a requirement. Using automatic or semi-automatic proof techniques, high confidence can be reached in the correct functionality. In particular, SAT-based model checking techniques [6], [7], [34] have been successfully applied to industrial scale hardware designs. However, their application is difficult and requires writing properties in temporal logics like LTL [30] or dedicated languages such as the *Property Specification Language* (PSL [1]).

In this article, we show how BDD can be extended to make it viable for hardware design. Instead of the original flow, which is based on acceptance tests only, we introduce a two-fold strategy, that also allows the user to write formal properties, which

1 Group of Computer Architecture, University of Bremen, Germany
2 LTCI, Télécom ParisTech, Université Paris-Saclay, Paris, France
3 Integrated Systems Laboratory, EPFL, Lausanne, Switzerland
4 CPS, DFKI Bremen, Germany
a) ulrich.kuhne@telecom-paristech.fr
b) mathias.soeken@epfl.ch
c) grosse@informatik.uni-bremen.de
d) drechsler@uni-bremen.de

This article summarizes and extends previous work published at *High Level Design Validation and Test Workshop* [12] and *Tests and Proofs* [11].

can be model-checked on the design. A major contribution is the seamless integration of formal methods into the BDD methodology: Maintaining the natural language style of the requirements, a unified specification document is created that is the source of both acceptance tests and formal properties. As a result, circuit components are created in an iterative fashion, while being tested and formally verified in a structured way, connecting the checked functionality to the natural language requirements.

Using the presented methodology, the best of the two worlds – static vs. dynamic verification – can be used. While test cases are often more intuitive for designers, properties can offer a much higher coverage and verification quality. Also, writing (and checking) a strong invariant in form of a property is often more concise than a collection of test cases witnessing this invariant. In terms of resource usage, simulating the design during regression tests is usually less time-consuming than running a model-checker on a complex design.

To fully benefit from the two-fold BDD strategy, we have created shortcuts between testing and formal verification. On the one hand side, we are able to generalise properties from the written test code. These properties are formally verified on the iteratively developed implementation and improve the coverage of the test cases. On the other hand, if a formal property fails, we can generate a debug scenario in natural language that describes the failing run in an intuitive way. This scenario can later serve as a regression test to verify that the specific corner case works correctly after changing the design.

To summarize, the main contribution of this work is a BDD flow for hardware design with the following properties:

- Test-driven BDD for hardware designs
- Executable textual properties similar to scenarios (Property-driven BDD)
- Automatically generalised properties from tests
- Automatically generated textual debug and witness scenarios from properties

Our approach is a first step towards productivity enhancements based on BDD for HW. Requirements and solutions evolve through collaboration and hence potential misunderstanding can be reduced. As a consequence the “correct” functionality is implemented and verified as correct, the costs are reduced while at the same time the quality improves.

This paper is structured as follows. Starting with the preliminaries in Section 2, the overall flow of the proposed methodology is presented in Section 3. The testing part is discussed in Section 4, while the integration of formal verification is presented in Section 5. Afterwards, Sections 6 and 7 show the interconnection of the two strategies, namely the generalisation of tests and the generation of debug scenarios, respectively. Related work is discussed in Section 8, before Section 9 concludes the work.

2. Preliminaries

2.1 Behaviour Driven Development

BDD is an extension of *Test Driven Development* (TDD) in which test cases are provided as a starting point and are the central elements along the whole design process. First, all test cases are specified, however, since no implementation is available, they will all fail initially. Guided by the error messages from the failing test cases, the implementation grows incrementally until all test cases eventually pass.

In BDD test cases are written in natural language rather than source code, offering a communication mean for both the designers and stakeholders. Test cases are called *acceptance tests*. They are structured by means of *features*, where each feature can contain several *scenarios*. Each scenario, in turn, constitutes one test case and is written following a *Given-When-Then* sentence structure. Each sentence is called a *step*. As an example, consider the following scenario:

Scenario: *Adding two numbers*
Given a calculator
When I add the numbers 4 and 5
Then I see the result 9 (1)

This scenario starts the implementation of a simple calculator and describes the addition of two numbers. In order to execute the scenario, the steps need to be bound to test code. This is achieved using so-called *step definitions* which are 3-tuples consisting of a keyword (which can be *Given*, *When*, *Then* or *And*), a regular expression, and the actual test code. All steps of a scenario are executed in the order in which they appear in the scenario. Whenever a step matches a regular expression, the corresponding test code is executed. The step definitions for the scenario described above are implemented as follows:

```
Given /^a calculator$/ do
  @calculator = Calculator.new
end

When /^I add the numbers (\d+) and (\d+)$/ do |a, b|
  @calculator.add(a.to_i, b.to_i)
end

Then /^I see the result (\d+)$/ do |a|
  @calculator.result.should == a.to_i
end
```

The example has been realized in Ruby [18] using Cucumber [40] as underlying tool to invoke the BDD flow. Note that the general flow can be applied to other programming languages and BDD tools accordingly. In Ruby, object instance variables are prefixed by an ‘@’ and functions such as ‘Given’ can get a block as parameter that is enclosed by ‘do’ and ‘end’. By making extensive use of operator overloading, *assertions* can be written intuitively such as in the last step definition using ‘should==’.

After defining the test code for each step, the scenario *Adding two numbers* can be translated to an acceptance test by the BDD tool. This is done by matching the steps with the regular expressions in the step definitions. During this process, parameters are being instantiated to form executable test code:

```
@calculator = Calculator.new
@calculator.add("4".to_i, "5".to_i)
@calculator.result.should == "9".to_i
```

Up to now, no implementation code has been written. While creating the step definition for the scenario, design decisions already have been made that affect the structure and the interfaces of the implementation. For example, it has been decided that the

calculator shall be realized as a class named *Calculator*, which currently has at least the methods *add* and *result*.

During the implementation phase, the test cases are usually executed whenever the implementation changes. The designer is guided by failing test cases, by *syntactic* errors – such as a missing implementation for a class or method – and *semantic* errors, which indicate failing assertions that need to be resolved. In our example, when executing the test cases, they will fail at the first step and stop with an error message stating that the name *Calculator* cannot be resolved.

Following the hints of the failing test cases, the designer needs to define a class and, subsequently, implement the methods *add* and *result* in a way that satisfies the acceptance tests. This ends the design process for this scenario. Using the BDD flow, the code is not tested as a post-process, but the tests are run during the whole implementation process, even before an implementation exists. In the successive steps, more features and scenarios can be added to complete the implementation of the overall system.

In order to improve the coverage of acceptance tests, feature files allow the specification of *scenario outlines* which provide parametrized scenarios enriched with *example tables* allowing for several test assignments. A scenario outline for the scenario in Eq. (1) for the pairs of addends $(2, 8)$, $(3, 9)$, and $(100, 20)$ can be written as follows:

Scenario Outline: *Adding two numbers*
Given a calculator
When I add the numbers <a> and <b>
Then I see the result <c>

Examples: (2)

a	b	c
2	8	10
3	9	12
100	20	120

2.2 Model Checking

During test-driven design of hardware or software, the quality of the resulting circuit or program is assured by directed tests. Each test case describes a particular use case in terms of inputs to the design under test, and defines the expected response or output. Testing then amounts to simulating the DUT and comparing the actual outcome with the values specified in the test case. In this way, it can be ensured that all relevant functionality is implemented and that at least one or several working use cases can be observed by the tests. However, it is practically impossible to *prove* the correctness of a design using tests, since it is infeasible to cover every possible computation of the DUT using simulation.

In many situations, model checking can provide a powerful alternative to simulation-based verification techniques. Based on a formal model of the DUT and a specification given in terms of some temporal logic, model checking proves that the design adheres to the specification. In the automata theoretic approach to model checking [10], [30], [38], the problem is reduced to checking if the language of the automaton composed of the system and the negation of the specification is empty. Unfortunately, the involved finite state machines can become very large, and the technique can only be applied to small systems.

The use of efficient data structures in symbolic model checking [8] has widened the application area model checking. Today, the most scalable solutions use Boolean satisfiability (SAT) solvers to search for counterexamples [5], [6]. Further improvements have been achieved using induction [7], [34] or abstraction and refinement [9].

In this work, we use a SAT-based inductive model checker similar to Ref. [34]. Given a safety property and a sequential circuit, the circuit is unrolled for k time frames and – together with the property – transformed to a SAT-problem. In this way, either a counterexample is found (disproving the property), or an inductive proof is attempted that the property always holds. If this cannot be proved, the length k is increased. The technique is implemented in the verification framework WoLFram [37].

2.3 Property Specification Language

In this work, we use the *Property Specification Language* (PSL [1]) to write assertions. PSL has been created in an effort to establish an industrial standard for assertions, which can be used for both dynamic and formal verification. Basically, PSL is a super-set of *Linear-time Temporal Logic* (LTL [30]), with an optional branching time extension. While PSL is a very rich standard, we only use a subset of its constructs, which is well suited to express the testing semantics of BDD. In particular, we concentrate on safety properties.

PSL is built on top of basic expressions in your favourite hardware description language (HDL), including Verilog. Assertions are formed by temporal operators and verification directives. As an example, consider the following simple PSL assertion:

```
assert always req -> next(ack);
```

Here, the verification directive `assert` tells the verification tool that the following expression should be checked on the design. The expression states that at each cycle in any computation (`always`), whenever the signal `req` is high, then (`->`) one cycle later (`next`), the signal `ack` should be high as well. A convenient way of expressing sequences of events in PSL are *sequentially extended regular expressions* (SEREs). Sequences are constructed using curly braces. Consider the following assertion:

```
assert always {req; ack} |=> busy;
```

Here, the sequence {req; ack} matches any computation where `req` is high and `ack` is high one cycle later. The overall temporal expression is assembled using the non-overlapping suffix implication (`|=>`). This means that whenever the sequence matches, then one cycle after the last cycle of the sequence, `busy` must be high. If the consequent expression should hold simultaneously to the last cycle of the sequence, the overlapping suffix implication (`|->`) can be used.

For a more detailed introduction to PSL, we refer the reader to Ref. [16]. The semantics of PSL and SEREs is formally defined in the appendix of the language standard [1].

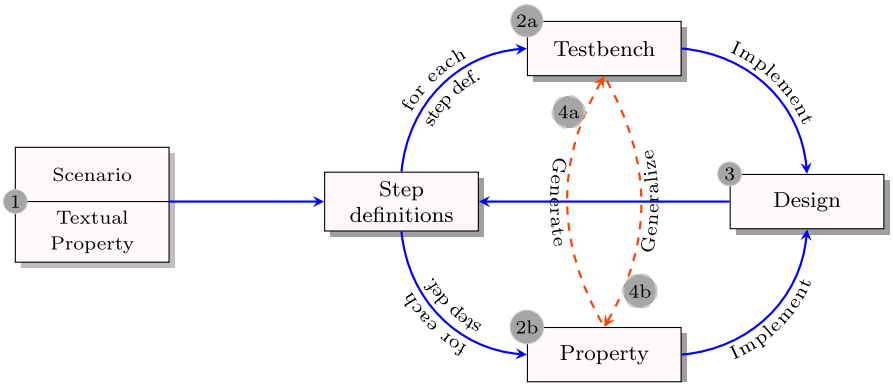


Fig. 1 BDD flow for hardware design.

3. Behaviour Driven Development for Hardware Design

The main contribution of this article is an extended BDD methodology, which has been adapted to the needs of a verification-centric hardware design flow. For this purpose, test-driven design is complemented by formal properties. Both test cases and properties are generated from a unified specification document. The overall flow is shown in **Fig. 1**. We will briefly sketch the involved steps in the following.

Before starting with the implementation itself, acceptance tests, essential temporal properties and invariants are captured in human-readable form (Step 1 in the figure). While this specification forms the basis for the subsequent implementation, it will keep evolving during the whole design process. In the next step, the described features will be implemented by the designer, supported by continuous testing and model checking runs. The upper half of the diagram in Fig. 1 represents the traditional BDD flow: Here, a testbench is created by assigning test code to each step (Step 2a). In the lower part of the figure, the steps are mapped to properties (Step 2b). Both tests and properties are guiding the designer in order to complete the design feature by feature (Step 3).

Although tests and properties are very different artifacts to begin with, there are useful ways of conversion between them. Test cases can often be generalized in order to cover a larger portion of the input space. These generalized test cases are in fact properties that can be verified by a model checker. This conversion is shown as Step 4a in the figure. In the opposite direction, scenarios can be extracted from a failing property. In this way, forgotten corner cases can be expressed in human-readable form as a textual scenario, and can be incorporated into the specification document as regression tests. This use case is shown as Step 4b in the flow.

The proposed hardware BDD flow will be described in detail in the following sections.

4. Testbench Specification and Timing

4.1 Running Examples

As running examples, an *Arithmetic-Logic Unit* (ALU) and a *First-In-First-Out* (FIFO) will be used. In **Fig. 2** the block diagrams of the example designs are depicted. The examples have been developed in Verilog. In Verilog a digital system is described as a set of modules. The individual modules can then be connected with nets for communication using ports. Both example designs are described using a single module as shown in the figure.

We use an ALU design as shown in Fig. 2 (a) from an open source hardware project [*1]. Next to the block diagram, part of the design's description is provided that shows more details on the port information. The ALU computes 17 2-input bitwise logic and arithmetic functions such as addition, shifting, multiplication and comparisons like equals or less. Using the single bit input `signed_i` it is indicated whether the two 32 bit data inputs (`a_i` and `b_i`) are to be treated a signed or unsigned integers.

While the ALU is a combinational logic circuit, the FIFO in Fig. 2 (b) is a sequential synchronous circuit that is driven by the clock signal `clk`. The FIFO has two operation modes: normal mode and bypass mode. In normal mode the FIFO can hold up to four elements. When the bypass mode is selected the FIFO is not operational; the FIFO does not buffer any input data. Elements are added and removed using the single input signals `push` and `pop`, respectively, where `dat_in` is used to present the data to be added to the queue. The single bit outputs `empty` and `full` provide information on the fill status of the FIFO, while `elems` gives the exact number of elements that are currently stored in the queue.

4.2 Testing Hardware Designs with BDD

In order to inspect a hardware design, a testbench is written that simulates the design under test. A testbench in Verilog is written as a separate module, as shown in **Fig. 3**. The testbench gives a test case for the FIFO design, where the total capacity of elements is inserted. The testbench module `fifo_tb` instantiates the FIFO design. The test case is declared in the `initial` block, which states a *behavioural* process that is only executed once. Also, a clock generator is described using an `always` block that continuously repeats its statements, thereby creating the synchronisation signal.

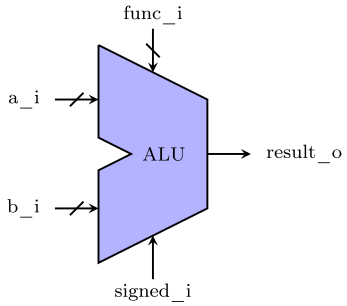
Since for hardware design such a separate testbench exists, the original BDD flow needs to be adjusted to handle this specific of hardware design. The starting point in the original and the new BDD flow is a failing scenario that is used to implement the intended behaviour of the design using test code from associated

[*1] http://opencores.org/project,m1_core

```
module m1_alu(
  input[31:0] a_i,
  input[31:0] b_i,
  input[4:0] func_i,
  input signed_i,
  output reg[32:0] result_o);

  always @(a_i or b_i
   or func_i or signed_i) begin
    case(func_i)
     ...
    endcase
  end

endmodule
```



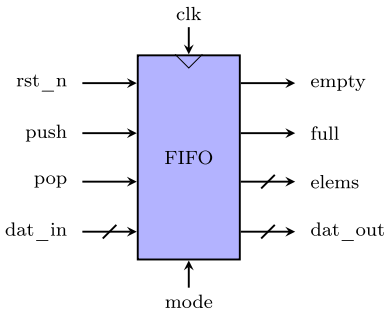
(a) Arithmetic-Logic Unit

```
module fifo(...);

  assign elems = full ? 4 :
                 (write_ptr - read_ptr) % 4;

  // Sequential behavior
  ...

  always @(posedge clk) begin
    if (reset) begin
      // Clear the fifo
      ...
    end
    else begin
      // Push and pop logic
      ...
    end
  end
endmodule // fifo
```



(b) First-In-First-Out queue

Fig. 2 Block diagrams of example circuits.

```
module fifo_tb;
   reg clk, rst_n, push, pop, mode;
   reg [7:0] dat_in;
   wire full, empty;
   wire [7:0] dat_out;
   wire [3:0] elems;
   integer i;

   fifo f( clk, rst_n, push, pop, mode, full, empty, elems, dat_in, dat_out );

   initial begin
    $monitor("dat_in %d - elems %d", dat_in, elems );
     rst_n <= 0;
     #2;
     rst_n <= 1;
     push <= 1;
     pop <= 0;
     for (i = 0 ; i < 5; i = i + 1) begin
       dat_in  <= i;
       #2;
     end
   end

   // clock gen and finalization
   initial clk <= 0;
   always  #1 clk <= !clk;
   initial #100 $finish;

endmodule
```

Fig. 3 Testbench for the FIFO example checking if the FIFO takes 4 elements.

step definitions.

Example 1. ***Figure 4*** *shows an example scenario that describes how data is written (push operation) to the FIFO from the previous section. When an element is pushed to the empty queue, then it is the oldest element in the FIFO and therefore it can be read from the output.*

After writing the scenario, step definitions (also shown in Fig. 4) are created and filled to generate test stimuli of the future testbench. All steps of a scenario yield a complete test case. To create a runnable test, all step definitions of a scenario are assembled to a single test case and added as an initial process of a testbench.

Another specific that is different for hardware designs is timing. Most designs have clocks that drive the design. In this case a clock generator needs to be added to the testbench. Since in a BDD flow all crucial behavioural information is included in the feature files, the clock tick information can also be added. This is done for all scenarios of a feature in the specification by us-

Scenario Outline: *Pushing*
Given the FIFO is operational
When the FIFO is empty
And I push <a>
And I wait 1 cycle
Then the output is <a>
Examples:

a
1
0
127

(a) Scenario

```
Given /^the FIFO is operational$/ do
  mode = 1;
end

When /^the FIFO is empty$/ do
  $assert( empty );
end

When /^I push (\d+)$/ do |arg|
  rst_n = 1;
  push = 1;
  pop  = 0;
  dat_in = arg;
end

Then /^the output is (\d+)$/ do |arg|
  $assert( dat_out == arg );
end
```

(b) Step definitions

Fig. 4 BDD scenario with step definitions.

ing the predefined background step "***Given*** *the clock (\w+) ticks every (\d+) time units*". The first parameter in this step definition names the clock of the module under test and the second gives the period of one clock cycle. A clock cycle is the time between two rising edges of the clock signal. We only consider single clock designs. A synchronisation delay can be described using the predefined step "**And** I wait (\d+) cycles?" (see Fig. 4) or directly in the test code using the delay operation of Verilog (e.g., *#3;*).

As briefly discussed in Section 3, *Then*-steps are usually used to describe assertions in a scenario. To support assertions in Verilog [*2], we made the `$assert`-function available for the BDD flow.

The body of the testbench can automatically be generated using the clock informations and the port information already described in the design's description or it can also be written by the designer manually as seen in the following example.

Example 2. *This example shows a user defined testbench for the FIFO example.*

```
module fifo_test;
  reg clk, rst_n, push, pop, mode;
  reg [7:0] dat_in;
  wire full, empty;
  wire [7:0] dat_out;
  wire [3:0] elems;

  fifo f( clk, rst_n, push, pop, mode, full, empty,
          elems, dat_in, dat_out );

  initial begin
    $yield;
  end
  // clock gen and finalisation
  initial clk <= 0;
  always  #1 clk <= !clk;
  initial #100 $finish;

endmodule
```

The '`$yield`' command is not a Verilog command but a placeholder which will be substituted with the corresponding Verilog test code from the individual scenarios. This testbench then needs to be made visible in the feature file specification by adding another background step: "***Given*** *the testbench (\w+)*", where the name of the testbench needs to be specified.

5. Property Specification

Properties are an essential part of a specification. They state expected behaviour by defining valid or invalid paths through the state space of the design. This section shows how properties can be defined in the BDD process as part of the feature specification. We call these properties *textual properties* and they are the counter part to scenarios, which are natural language test cases. Textual properties are defined alongside scenarios as a second driver for the development of the design. Like scenarios, also textual properties contain of one or more steps. The main idea is to reuse the existing semantics that are used for describing properties to describe textual properties.

There are some differences in the application of properties and scenarios. While scenarios directly translate to test cases that are run in a testbench, textual properties need to be transcribed to formal properties that can be checked using an automatic verification tool. Consequently, step definitions of textual properties do not contain test code but are written in a property specification language (such as PSL).

A textual property can be specified in two different ways:

(1) as an *implication property* where steps are given in terms of the *Given-When-Then* structure (known from scenarios), or
(2) as an *invariant property* which consists of a single step and no *Given-When-Then* keyword.

Both types are described in more detail in the following subsections. Please note that even if in the beginning simple (bit-level) expressions are used when formally capturing the behaviors, our approach fully supports word-level specification in the step definitions.

5.1 Textual Implication Properties

A special aspect of a scenario is its structural semantics that is very similar to a formal property. The *Given-When-Then*-structure translates very nicely to the formal setting.

Example 3. *An implication $p \rightarrow q$ can be expressed in natural language with "*When [*3] *p is true,* then *q needs to be true, too." The first part — starting with the keyword* When *— is used to describe the antecedent p while the consequent q of the property is indicated by a* Then *keyword.*

The semantic structure of the property in the example is very similar to the structure that is used for most scenarios. Using the existing keywords *When* and *Then* it is possible to describe an implication property using the existing structural semantics of scenarios.

[*2] Our implementation builds on Verilog IEEE Std. 1346-2001

[*3] Although *If* would be semantically correct, *When* can be used as synonym.

In order to create a property that can be used for formal verification, the textual property needs to be assigned to a correct property in PSL. The step definition code of all *When*-steps of a scenario belongs to the antecedent, the consequent is filled with the step definition code of all *Then*-steps of scenario. In this way, the verification intent of the test scenario is captured in a PSL property.

The *Given*-keyword is usually used to set up the context for a scenario [40]. This is similar to textual properties where global assumptions – such as configurations or environment constraints – need to be defined.

The following example illustrates how to specify textual implication properties in the BDD process.

Example 4. ***Figure 5*** *(a) shows an implication property that states that the number of elements of a FIFO is increased whenever an element is pushed. This property is only valid if the FIFO operates in normal mode, which is defined in the* Given*-step. The property looks very similar to a scenario which would be used for testing, but instead of assigning test code to the steps, the desired behaviour is expressed with PSL code. The step definition code in PSL is given in Fig. 5 (b).*

The PSL property code is written in a Verilog flavour. To build the property, the designer specifies for which part of the property, i.e., antecedent, consequent, or assume, the given PSL code is written. For each part an API command is provided.

Although this information could in principle be generated from the appropriate keywords (*When* as antecedents, *Then* as consequents, and *Given* as assumes), it is not generated automatically, so that properties can be written more flexible. In some cases those keywords are not suitable at all as described in the following section.

Since step definition can be reused in every part of the property (antecedent, consequent or assume part), the specified API commands in the step definitions are checked against the used keywords in the scenario. If they mismatch, a warning is displayed and the step definition code is inserted to the property part defined by the keyword since the natural language description are given a higher ranking.

Property: *Incrementing*
Given the FIFO is in OP mode
When the FIFO is not full
And I push an element
And I wait 1 cycle
Then the number of elements has increased

(a) Property

```
Given /^the FIFO is in OP mode$/ do
Verilog::add_assume do
  mode == 1
end
end

When /^the FIFO is not full$/ do
Verilog::add_antecedent do
  !full
end
end

When /^I push an element$/ do
Verilog::add_antecedent do
  rst_n == 1 && push == 1
  && pop == 0
end
end

Then /^the number of elements
has increased$/ do
Verilog::add_consequent do
  elems == prev(elems) + 1
end
end
```

(b) Step definitions

Fig. 5 Implication property.

5.2 Textual Invariant Properties

Invariant properties describe global requirements of a design that cannot be easily stated as a set of scenarios. An invariant describes *safety* behaviour which means that a specific state should never be reached [26]. For instance, two signals `read_enable` and `write_enable` cannot be asserted at the same time or a specific capacity cannot be exceeded. For an invariant property an implication structure is not applicable. Hence, the usual *Given-When-Then* structure may be superfluous.

Example 5. *The following property verifies the maximum amount of elements that can be stored in the FIFO.*

Property: *Invariant*

** The number of elements in the FIFO is at most 4.*

The following step definition contains the PSL code for the property, stating that the number of elements shown at output `elems` *will at most be 4:*

```
Then /^the number of elements in the FIFO is at most 4\.$/ do
Verilog::add_consequent do
  elems <= 4
end
end
```

Both descriptions can be viewed as natural language patterns for describing properties. Property specification patterns haven been introduced by Dwyer et al. in Ref. [14] where the authors proposed patterns for commonly occurring requirements. The implication properties can be viewed as subset of the Response patterns. Invariant properties can be used to describe properties from the Absence pattern.

5.3 Assembling the Property

The property code in the step definitions needs to be assembled to a correct property in PSL syntax in order to be checked against the implementation. The structural semantics of the textual property defined by the keywords and the API calls is used to map the appropriate parts to the property. The antecedent parts (also given by *When*) and the consequent parts (also given by the *Then* keyword) of the step definitions is mapped to the antecedent and the consequent of the resulting property, as can be seen in **Fig. 6**.

The PSL code of each step is joined for the antecedent and consequent block, respectively. If the property code of the antecedent (respectively consequent) occurs in the same time step, they are assembled as parallel sequences using the *non-length matching and* '`&`' operator. Steps in consecutive cycles are treated using the concatenation operator '`;`' between the statements.

In the textual property of Fig. 6 (a), timing is explicitly stated with the predefined step *When I wait 1 cycle* which separates the antecedent and consequent blocks. Using the non-overlapping suffix implication operator '`|=>`' in the generated property, it is expressed that the consequent is expected to hold one cycle after the last cycle of the antecedent. The transcription of timing to the PSL property is depicted in the figure using a dashed arrow.

The last step definition is special since it defines an *assume*-block which describe global restrictions. As a result, an independent property is assembled for every *assume*-block given in a property. In the example, the configuration of the FIFO — the

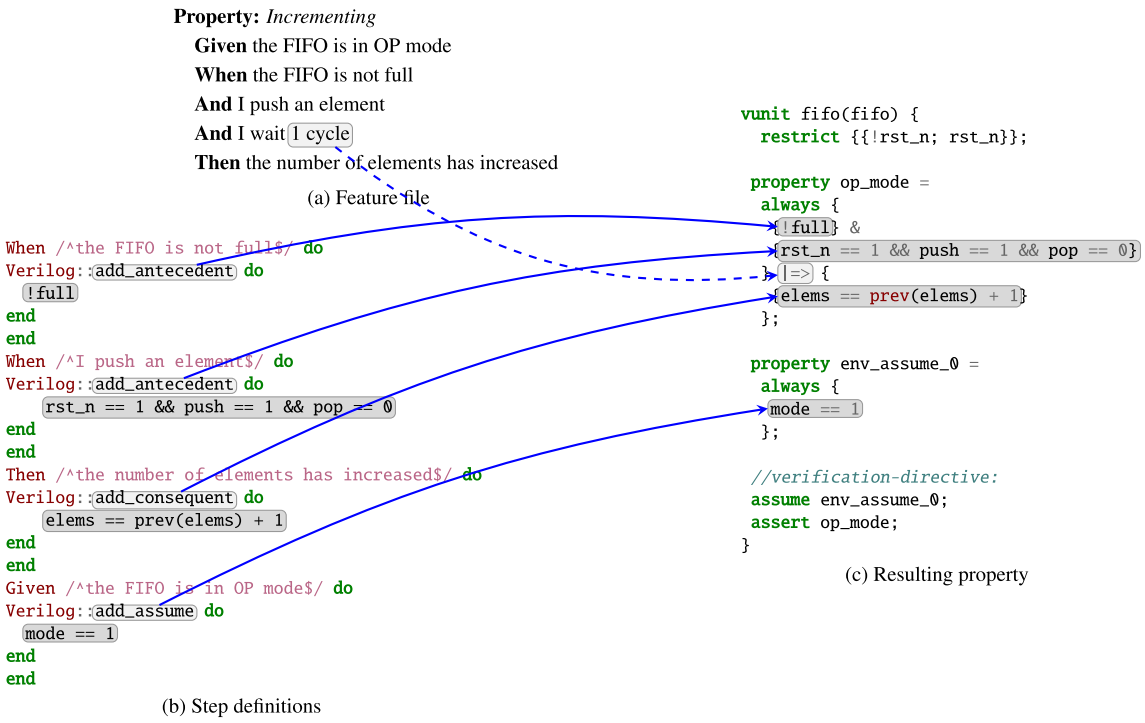


Fig. 6 Implementation.

FIFO operates in normal mode — is constrained. Then this property is *assumed* in the context of this PSL property.

The idea shown in this section leverages the experience of system designers when writing properties for the first time. A designer can start with easier PSL construct with just simple LTL, which can be seen as simple if conditions of standard HDL code.

6. Testcase Generalisation

Specifying textual properties as a second driver for the implementation helps to increase the confidence in the correctness of the DUV. But, in the design process it is often more natural and easier to write a test case than specifying a property. Hence, there might exist scenarios that could have been easily stated as properties in order to cover the complete input space of the scenario, while by using scenario outlines only a subset of test patterns are applied.

In the previous section we have shown how to write properties in the same manner as natural language scenarios and drive the implementation using these properties. Scenarios are very similar to the specification of implication properties since they build on the same structural semantics. In the following, we will show how to use these scenarios to automatically generalise PSL properties from them and use them for formal verification.

Example 6. *Consider the scenario outline in* ***Fig. 7****, which describes pushing to an empty FIFO: After insertion of an element, it will be visible at the output in the next cycle. Using the example table construct, three tests will be run with different values for the input element.*

Scenarios usually consider only few selected test input data and never cover a scenario exhaustively. The example in Fig. 7 only covers the inputs 1, 0 and 127 explicitly. Covering the whole input space would require an explicit enumeration in the examples table, which is obviously infeasible for larger bit widths. In the same time, the usage of <a> as a placeholder already suggests using a model-checker instead, leaving the data as a free input.

Scenario Outline: *Pushing*
Given the FIFO is in OP mode
When the FIFO is empty
And I push <a>
And I wait 1 cycle
Then the output is <a>
Examples:

a
1
0
127

Fig. 7 Natural language scenario with poor coverage.

Both parts of the specification, textual properties and scenarios have the same validation intent which is described using the same characteristic structural semantics given by the *Given-When-Then*-structure. Hence, such scenarios (in particular the scenario outlines) are especially well suited for generalisation in terms of a PSL property. This property can then cover the whole test input space. To obtain the PSL property, the structure of the scenario defined by the *Given-When-Then*-keywords is mapped to an implication property. Especially the When-Then semantic structure can be mapped directly to an implication, as shown in Section 5.

A property is obtained by mapping the structure of the scenario determined by the *Given-When-Then*-keywords and the underlying test code to an implication property. While the code of all *When*-steps of a scenario constitute the antecedent of a property, the code of all *Then*-steps constitute the consequent, thereby capturing the verification intent of the scenario in a PSL property.

The following algorithm explains how properties are generated from scenarios.

Algorithm P (*Property Generation*). Given a scenario and its step definitions, this algorithm generates a property for it.

P1. [Assigning step definitions.] For each step in the scenario, the step definition code is added to a skeleton property. The position of the code depends on the keyword: *Given* steps are assumptions, *When* steps are part of the antecedent, and *Then* steps are part of the consequent.

P2. [Resolve dependencies.] Since inputs and outputs need to be related, the parameters used to set the test input data inside the step definition must be replaced by the placeholder variable from the scenario.

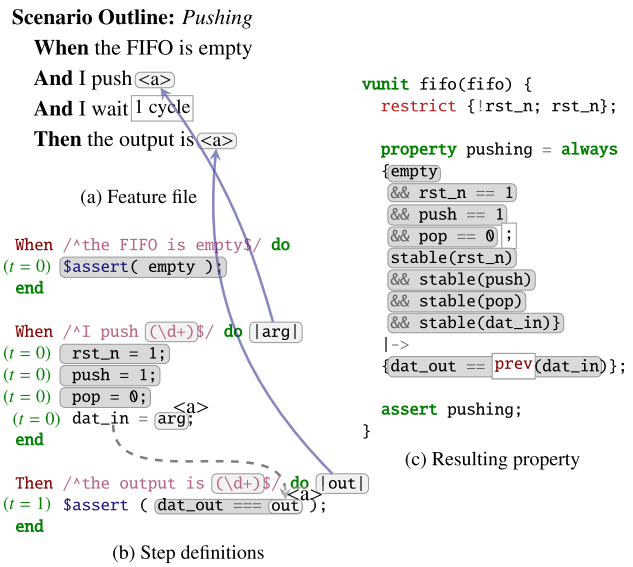
P3. [Timing.] Timing information from all step definitions is extracted and each statement of the step definition is assigned to one time step.

P4. [Test semantics.] In order to follow the same semantics as in the test, the property is extended by expressions that ensure the test semantics: Each attribute that does not change over time is fixed using the `stable` macro which constraints an attribute's next value to equal its current value.

P5. [Assembling.] Assemble all statements of the antecedent and the consequent to SERE expression using the timing information of step P3 and the additional test expressions of step P4.

The algorithm P is illustrated in **Fig. 8**. As described in P1, all statements in Fig. 8 (b) with a grey background are sorted to the proper part of the implication property. Depending on the keyword that is used in the scenario (see Fig. 8 (a)), a statement will be inserted to the antecedent (if it is a *When*-step) or inserted to the consequent (if it is a *Then*-step). Statements that only set test input data to an input are skipped in this step, therefore the last assignment of the second step definition is not included in the antecedent of the property.

After that the dependencies between the inputs and the outputs are resolved. For this purpose, the implicit information of the *glue code* is used. The glue code is the part of the scenario and the step definitions that relates the input and output signals with placeholders such as <a>. Placeholders correspond to selected test input data. Since the same placeholder variables are used to target the same inputs and outputs in the scenario, it is possible to resolve the dependencies in the step code. In Fig. 8 (b) the parameters `arg` and `out` are substituted by the placeholder <a>. This is indicated by the solid arrows that connect the parameters of the step definitions with the placeholder <a> in the scenario. Both mark the same input `dat_in`. For this reason, the parameter `out` can be replaced by the input `dat_in` in the last step definition, which is indicated by the dashed arrow.



(a) Feature file

(b) Step definitions

(c) Resulting property

Fig. 8 From a scenario to a generalised property.

6.1 Limitations of Property Generalisation

The generalisation of scenarios to properties allows to improve the verification coverage without any user intervention. The only overhead is in terms of runtime when applying model checking instead of mere testing. When formal methods are applicable to a design, then the generated properties can cover the whole input space and can potentially discover more bugs. This makes the generalisation very attractive. However, generalising simple test cannot fully replace a full fledged property suite, and some design intent that is well captured by complex temporal properties will not be generalisable from user written scenarios. One possibility to overcome the problem is to generalise a property from several test cases and not only one, as more data can be gathered this way. This also avoids having a large amount of fine-grained small properties.

Another pitfall that frequently occurs in scenario outlines is functionality implicitly described by example data. Consider again the scenario outline in Eq. (2). There, the intended functionality (adding two numbers) is given by the examples that define the correct output <c> for inputs <a> and <b>. In the step definitions of the scenario, there is no hint whatsoever about the semantics of the tested function. Therefore, generalising this scenario will not result in a useful property that verifies the addition. A similar problem occurs if environment assumptions are implicitly given by the test data. Such assumptions are e.g., restrictions on the data range of certain inputs, special relationships between inputs or even specific temporal sequences of inputs.

As a rule of thumb, in order to avoid such pitfalls, outputs should not be constrained by means of an example table, if possible. Instead, explicitly stating the relationship between the inputs and outputs in the scenario allows generalization. As for the adder example, the scenario could be improved by providing a step that precisely describes the expected output to be the sum of <a> and <b>. In the case of environment constraints, these can often be stated as a *Given*-sentence. These rules also help to improve the quality of the scenarios when serving as a documentation of the DUV.

7. Generation of Debug and Witness Scenarios from Properties

To simplify the utilisation of properties furthermore in a BDD manner, this approach generates exemplary scenarios for properties, either when the property check fails such that the developer receives a *debug scenario* for finding the bug in the DUV or as an explanation for any written property in terms of a *witness scenario*. The idea of this approach is to generate executable scenarios that can be understood and reused by the developer for further

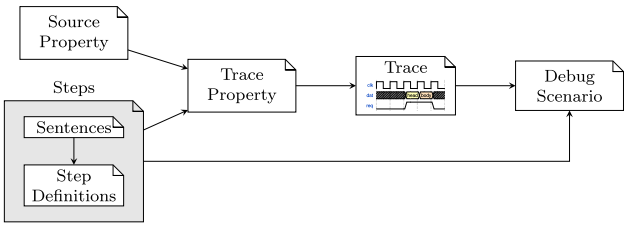


Fig. 9 Scenario generation overview.

investigations of the design. Therefore, this approach generates a debug scenario or a witness scenario for a property by reusing existing steps that the developer already used in other scenarios.

In order to find a witness scenario for any valid property or a debug scenario for an invalid property, a *trace property* needs to be created that considers the existing steps for the creation of a counterexample or witness trace.

Figure 9 shows the general outline of this approach. To be able to form a trace property, the source property and all executable step definitions have to be known. The counterexample, which is generated through bounded model checking, is transformed to a new scenario consisting of — as much as possible — existing steps. To retrieve the known steps for a new scenario, all steps need to be encoded in the trace property, so that they can be read from the counterexample.

Algorithm S (*Scenario Generation*). Given a property p and all already written scenario steps and their step definitions, this algorithm generates a set of good exemplary scenarios for it.

S1. [Pre-process scenario steps.] As a pre-process, all steps together with their corresponding step definition code need to be gathered. This is done while all scenarios are simulated. Since the step definition code contains only Verilog code these steps need to be generalized similar to the approach in Section 6.

S2. [Trace property.] Using the property and all generalized steps, a trace property is formed to find a scenario that generates a debug (or witness) trace for the source property.

S3. [Trace scenario.] After the model check is executed, each trace of the counterexample is decoded to form a BDD scenario.

S4. [Find good generated scenarios] Evaluate all scenarios using optimisation techniques to find the best possible scenarios for the developer.

While Algorithm S only outlines the general method of the scenario generation approach, the following subsections will explain the individual steps in more detail.

7.1 Pre-process Scenario Steps (S1)

In order to create a debug or witness scenario for a (failing) property, the counterexample trace which results from a failing model check needs to reference the existing steps in the debug or witness trace. This information is then used to create a new scenario. A step is only relevant if it is associated with step definition code in Verilog that can be executed without any (syntax) errors since the resulting scenario shall be executable. Hence, all steps and their correct step definition code are gathered during simulation. To integrate these step definition into the trace property, each step definition code needs to be translated to PSL, which can be done using the approach from Section 6.

7.2 Creating a Trace Property (S2)

The trace property intends to generate a trace for the (failing) source property. Instead of generating an arbitrary counterexample, this trace shall consist of as many existing steps as possible, so that an easily comprehensible scenario can be put together. To achieve this, two technicalities need to be encoded into the trace property: all runnable steps and the verification intent of the source property. Depending on the former property checking result (if any) it needs to be handled differently. At first we will introduce the generation of a trace property that will result in a new scenario for debugging, which is also shown in **Fig. 10**.

Figure 10 displays an example for a generated trace property for a property that has failed during the property check. The property in the figure describes the amount of elements that are contained in the FIFO if the FIFO is full. This property will be used as a running example in this section. We made this property fail by inserting a bug into the FIFO design which returns a wrong number of elements contained in the FIFO. The error has not been covered by any user defined scenarios.

To find a counterexample for the failing source property shown in Fig. 10 (a), the reversed verification intent of the property needs to be encoded to find a counterexample for the original verification intent. The second line of block 2 shows how to encode an implication property. Block 1 illustrates that every step must be encoded in the trace property. The scenario created from the counterexample should be composed only of steps, hence, at least one step needs to hold which is enforced by the first line of block 2.

The described kind of property generates a counterexample that can be used to create a new scenario for debugging the failing property. In case a witness scenario shall be generated, the constraint in block 2 would be created as follows

```
property full_fifo = never (antecendent && consequent);
```

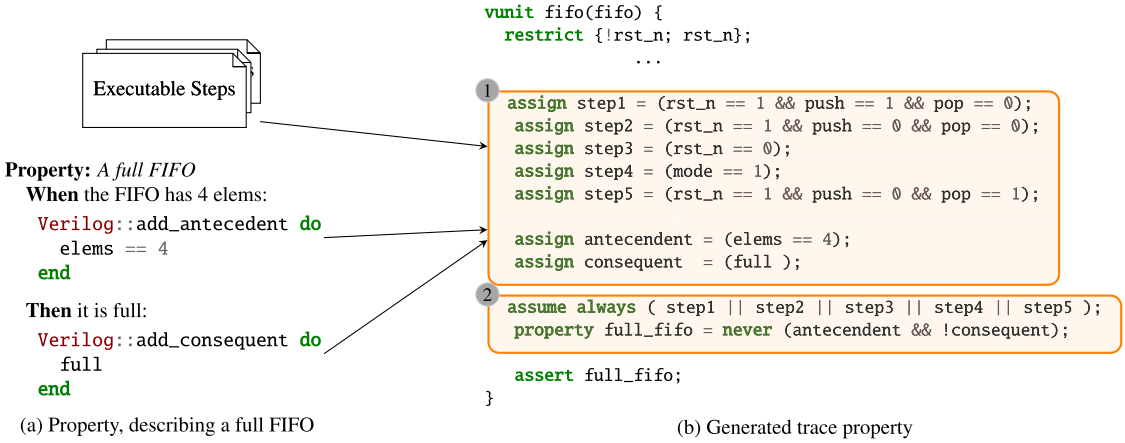
In this case we want a witness for the valid property, therefore the consequent can not be negated in order to find a trace containing the correct verification intent.

If the property is an invariant the antecedent clause is left out for both kinds of trace properties as shown in the following constraint

```
property full_fifo = never (consequent);
```

In order to give a better explanation for a debug (or witness) trace different kinds of traces are created. This is done in two different ways: (1) allowing clock cycles without user steps and (2) banning existing traces.

For the first way a trace property is created that requires less known steps in the resulting scenario. This is done by allowing to set DUV inputs independent of encoded testing steps in some time steps, which is achieved using e.g., a modified assume directive (block 4) such as the following:



(a) Property, describing a full FIFO

(b) Generated trace property

Fig. 10 Generating a witness property for an implication property, that describes when the FIFO is full.

```
assume always next_e[0..1] ( step1 || step2 || step3 ||
                             step4 );
```

In this implementation only every second time step, a predefined step is assumed to be part of the trace. Therefore, in between these steps, a behaviour that is not representable by known steps is allowed in the trace.

Additionally it can be useful to restrict the counterexamples by prohibiting known traces. Considering the example above, this is done by adding the following constraint to the property:

```
assume never {step1; step2};
```

This excludes any sequence containing of `step1` followed by `step2`. Although this is rather strict, it can generate considerably different scenarios which enable a different view on the design.

7.3 Creating a New Scenario (S3)

After the model checker has generated a counterexample, it needs to be converted into a runnable scenario. The counterexample consists of assigned input signals for every time step. The encoded steps in the property appear as assigned signals in the counter example. A step signal that is assigned in a specific time step can be directly converted to a natural language step. All transformed steps result in a new scenario. Nevertheless, it may be possible that an assigned step does not cover all input signals of the DUV, therefore the counterexample may contain additional relevant input assignments. These assignments also need to be converted to new steps. Furthermore the assertion needs to be encoded into a new *Then*-step, too. As a result, we categorize the creatable types of steps as follows:

***User step*:** A *user step* is a step that has been defined by the developer for writing tests. It is encoded into the trace property and can directly be used in the new scenario, when it is assigned in the counterexample.

***Assign step*:** Since some input signals can be assigned in the counterexample that are not covered by any user steps, these assignments need to be encoded as new, self-contained steps. These steps are called *assign steps*. For example a step, which sets the input signal `dat_in` to `42` will be named "**And** `dat_in` is set to 42".

***Check step*:** To express the verification intent of the failing property in the new scenario, *check steps* are created. They contain assertions and often start with the *Then*-keyword. Check steps are added at the end of a new scenario. The creation of these steps from PSL code is explained in detail below.

***Aux step*:** An *aux step* is a helper step that eases the creation of check steps. Its application and creation is described in the following subsection in more detail.

7.3.1 Generating Check Steps

To check the verification intent of the failing property at the end of the newly generated scenario, the property needs to be transformed to assertion test code and added as *Then*-steps. Two types of properties are considered here: implication and invariant properties. While an implication property consists of two check steps, the first represents antecedent and the second the consequent of the property, an invariant property only consists of one check step.

Since a property may be expressed over several clock cycles, everything that takes effect in a separate clock cycle needs to be encoded apart from other cycles and inserted in the right time step of the new scenario.

Example 7. *Consider the following sequence of an antecedent that will be represented by a check step*

```
{ empty ; push==1 }
```

This sequence can be effectively checked by the following test code

```
check_reg_1 = empty;
#2;
check_reg_2 = push == 1;
$assert(check_reg_1 && check_reg_2);
```

The individual assignments of each cycle are saved in a register at different time steps. This enables the complete sequence to be checked in the last time step.

Since the registers used in the sequence need to be assigned in the new scenario, a special natural language step, called *aux step* is generated, that makes the technical overhead visible to the developer. The step for the first assignment saved in "check_reg_1" is named "**And** the check reg 'check_reg_1' is set to `empty`".

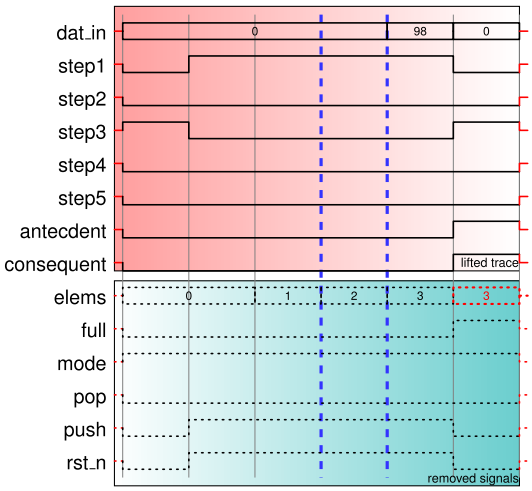


Fig. 11 Generating a scenario from a counterexample.

7.3.2 Translating the Counterexample into an Executable Scenario

Before translating the counterexample to an executable scenario it is necessary to have a counterexample as concise as possible. Using approaches such as Ref. [31] the counterexample can be reduced such that it only contains a minimal (relevant) set of assignments. This reduced counterexample is then translated to a trace scenario. After applying this technique a concise counterexample is extracted where each assignment to step and input signals is translated to user steps and assign steps, respectively. The verification intent of the property is translated to aux and check steps according to the method described above.

Consider the trace shown in **Fig. 11** for the example property in the previous section. The top eight signals are left after the assignments of the counterexamples have been lifted. The signals below have been reduced but are kept in the trace for the sake of completeness. The trace of the counterexample consists of six time steps.

The translated scenario of this trace is shown in **Fig. 12**. The dashed lines in the trace of Fig. 11 highlight the fourth time step, where an element ("0") is pushed to the FIFO, which is also described by the first step (encoded with `step1`) named *I push 20*. In the resulting scenario on the left hand side of Fig. 12 the time steps 2 to 5 contain an assignment step that is redundant since both steps set the `dat_in` signal but only the last value is important. This redundancy occurs due to the placeholder of the step definition that is set to "20" in the original step but the corresponding input signal is set to "0" in the counterexample. Hence, the placeholder value is replaced such that the assigned value in the counterexample is used. This is shown on the right hand side of Fig. 12 where the assignment information of the input signal `dat_in` is merged into the step *I push 0*.

7.4 Find Good Generated Scenarios (S4)

Steps 1 to 3 generate a large number of scenarios that the developer could use for debugging the failing property. Selecting the best scenarios can be hard. This section proposes an approach to find a set of good scenarios from which the developer can choose.

In order to detect a good scenario, we need to define what makes a scenario a useful supplement to the test suite. In fact, there are several aspects, some of which conflict with each other: On the one hand side, a scenario should be as concise as possible, since shorter scenarios are easier to understand for a human reader and since they consume less simulation time. On the other hand, a good scenario should have high code coverage to be useful. Some corner case related statements in the code might be hard to reach, therefore requiring long and complex scenarios. Finding "good" scenarios is a *multi-objective optimization* (MOO) problem, and we will use MOO techniques to solve it. Before giving the exact criteria for the problem at hand, we will introduce the central notions of MOO in the following. In particular, we are interested in *Pareto optimal* solutions, for which none of the objectives can be improved without compromising one or more of the other objectives.

Definition 1 (Pareto optimum)**.** Given the solution space Ω, an evaluation function $f : \Omega \to \mathbb{R}^n$ maps a solution to a vector of n objectives. For each dimension i, an ordering relation $\prec_i \in \{<, >\}$ is given, depending on whether objective i is minimized or maximized, respectively. A solution p *dominates* another solution q if the following holds:

$$p \prec q \;\Leftrightarrow\; \exists i : f_i(p) \prec_i f_i(q) \wedge \nexists j : f_j(q) \prec_j f_j(p) \tag{3}$$

The *Pareto set* $\mathcal{P}$ is then defined as the set of all non-dominated solutions:

$$\mathcal{P} = \{p \mid \nexists q \in \Omega : q \prec p\} \tag{4}$$

In order to apply this notion in the context of scenario generation, we need to define the optimization criteria. At first, we will review the definition of a scenario in terms of automatic scenario generation including the different types of steps presented in the previous section.

Definition 2 (Automatically generated scenario)**.** Let $S = \langle s_1, \cdots, s_n \rangle$ be a scenario with n steps. Each step can have one of four types: $s_i \in U_S \cup A_S \cup C_S \cup X_S$, where U_S is the set of all *user steps*, A_S is the set of all *assign steps*, C_S is the set of all *check steps* and X_S is the set of all *aux steps*. The length of S is defined as $len(S) = n$.

For a generated scenario to be useful, we want it to be as short as possible. Furthermore, in order to obtain a readable scenario that the user can easily make sense of, it should ideally consist of user steps only, since these steps have previously been defined by the user. This gives rise to the first two objectives: Minimizing the length, while keeping the ratio of user steps high. The remaining dimensions are made up by coverage metrics.

Definition 3 (Coverage)**.** The coverage $cov(S) = (c_1, \ldots, c_n)$ of a scenario S is an ordered n-tuple where n is the number of coverage criteria of that can be measured running S separately by the underlying coverage tool and $c \in [0..1]$.

Obviously, the coverage of a scenario should be as high as possible. In our implementation we use the tool `covered`[*4] which calculates six coverage metrics: line coverage, toggle coverage, memory coverage, combinational logic coverage, FSM coverage and assertion coverage. Overall, the objective function is defined

[*4] http://covered.sourceforge.net/

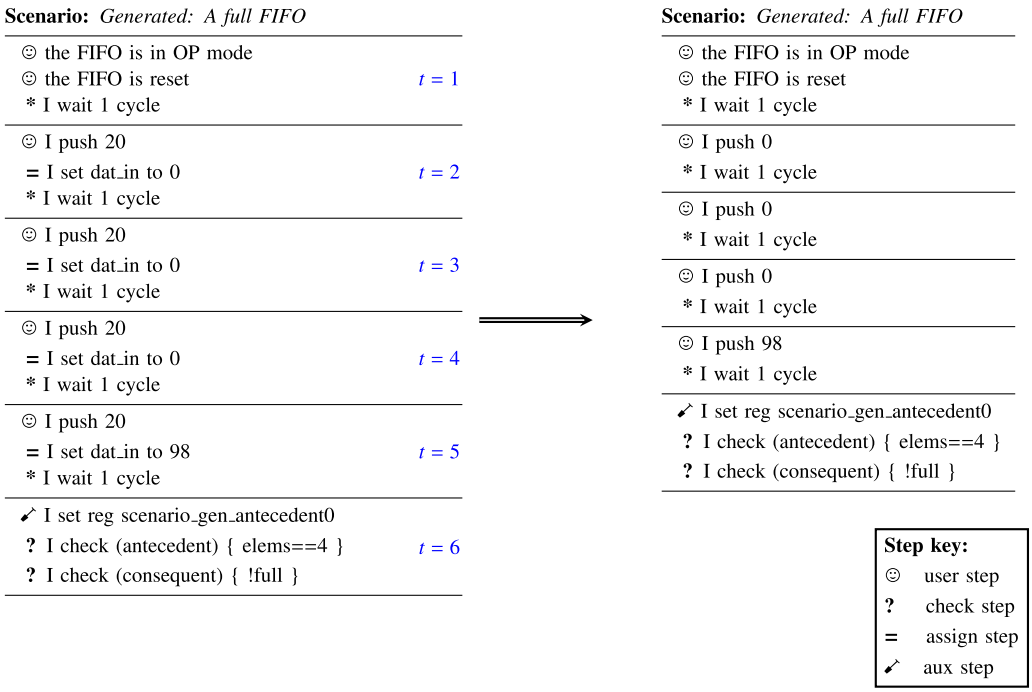


Fig. 12 Debug scenario for the failing property *A full FIFO*.

as follows.

Definition 4. Given the set of all generated scenarios Ω, the objective function $f : \Omega \to \mathbb{R}^8$ is given by

- $f_1(S) = len(S)$ with $<_1$ defined as $<$
  (Get a scenario that is as short as possible)
- $f_2(S) = \frac{|U_S|}{len(S)}$ with $<_2$ defined as $>$
  (Get as many user steps in the scenario as possible)
- $(f_3(S), \ldots, f_8(S)) = cov(S)$ with $<_i$ defined as $>$ for $i = 3 \ldots 8$
  (Get the highest possible coverage)

Example 8. *Consider the generated scenarios for the failing property* A full FIFO *in* ***Fig. 13****. The first generated scenario is the longest scenario, it contains the most user defined steps and has the highest code coverage, since it also applies the* pop *operation. But this additional operation does not suit the semantic meaning of the new scenario. The FIFO is filled to be fully stocked; a removal of elements is counter-productive. The second and third scenario have the same code coverage but differ in the number of user defined steps. While the second scenario only consists of mostly user steps, the third scenario uses mostly assign steps. Although the third scenario is more concise, it hides the behaviour of the test case and therefore is not considered as a good scenario.*

8. Related Work

8.1 Specifying Properties

This section reviews the diverse related work relevant to the application of formal properties in an agile development flow. Two research fields can be distinguished for the application of properties in an agile framework such as BDD: combining formal methods with agile development processes and easing the usability of formal verification.

8.1.1 Formal Methods in Agile Development

The combination of formal techniques and agile development has been considered in Ref. [22]. There, Henzinger et al. propose a paradigm called “extreme model checking”, where a model checker is used incrementally during the development of software programs. Another approach is called “extreme formal modeling” [36]. In contrast to our work, a formal model is derived first, which can then be used as a reference in the implementation process. The technique has also been applied to hardware [35].

The work by Nummenmaa et al. [28] describes an approach that uses an executable formal specification in an agile development process. The authors combine a formal specification with realistic user interaction, thereby visualizing formal methods.

8.1.2 Easing the Usability of Formal Methods

The work in Refs. [14], [15] proposes a first approach to ease the creation of formal specification by introducing a property specification pattern system that generalises descriptions of commonly occurring requirements. Parts of this pattern system is applied in the property-based BDD of this work.

The authors of Ref. [33] use a FSM notation to add formal verification to the verification process with little to no knowledge of formal methods. This work also eases the use of formal verification by reducing the new training to a minimum by using natural language to approach formal verification step by step.

8.2 Generalising Properties from Tests

In Ref. [2], Baumeister proposed an approach to generalize tests to a formal specification. His work considers Java as target language where the specification is checked using generated JML constraints. The drawback is that the approach does not facilitate an automatic generalization of tests.

A property-driven development approach is presented in Ref. [3], where a UML model is developed together with a specification and tests in a TDD manner and OCL constraints are being added to the UML models while generalizing test cases. How-

Scenario: *Generated: A full FIFO*

☺ the FIFO is in OP mode
☺ the FIFO is reset
* I wait 1 cycle

☺ I push 0
* I wait 1 cycle

☺ I pop an element
* I wait 1 cycle

☺ I push 0
* I wait 1 cycle

☺ I push 0
* I wait 1 cycle

☺ I push 0
* I wait 1 cycle

☺ I push 98
* I wait 1 cycle

🖌 I set reg scenario_gen_antecedent0
? I check (antecedent) { elems==4 }
? I check (consequent) { !full }

(a) Scenario 1

Scenario: *Generated: A full FIFO*

☺ the FIFO is in OP mode
☺ the FIFO is reset
* I wait 1 cycle

☺ I push 0
* I wait 1 cycle

☺ I push 0
* I wait 1 cycle

☺ I push 0
* I wait 1 cycle

☺ I push 98
* I wait 1 cycle

🖌 I set reg scenario_gen_antecedent0
? I check (antecedent) { elems==4 }
? I check (consequent) { !full }

(b) Scenario 2

Scenario: *Generated: A full FIFO*

☺ the FIFO is in OP mode
☺ the FIFO is reset
= I set pop to 0
= I set dat_in to 0
* I wait 1 cycle

= I set push to 1
= I set rst_n to 0
* I wait 1 cycle

* I wait 1 cycle

* I wait 1 cycle

= I set dat_in to 12
* I wait 1 cycle

🖌 I set reg scenario_gen_antecedent0
? I check (antecedent) { elems==4 }
? I check (consequent) { !full }

(c) Scenario 3

Step key:
☺ user step
? check step
= assign step
🖌 aux step

Fig. 13 Selecting good scenarios.

ever, this approach is not implemented.

Automatic generation of properties has been described in Ref. [23]. However, in that work, properties and constraints are generated from *Production Based Specification*, which is a formal specification language based on regular expressions.

8.3 Generating Scenarios from Properties

Two research fields can be distinguished for the application of properties in an agile framework such as BDD: test cases generation from formal properties and finding good test cases.

8.3.1 Test Cases from Properties

Visser et al. presented an approach in Ref. [39] that generate test inputs using model checking and symbolic execution. In this work test inputs are computed for Java white box tests. The counterexamples to the properties are used to report the test inputs. This is similar to our work with the differences that this approach generates natural language scenarios that can be used for hardware testing.

In Ref. [4], the authors extended the software model checker `Blast` such that it automatically generates test suites with a full coverage according to a given predicate for C programs. This approach check if a specific line of code is reachable and returns a test case for the line. In contrast, our approach allows different kinds of properties. Furthermore the solution presented in this work is applicable for hardware designs.

In the work [29] by Oberkönig, Schickel and Eveking a method has been presented that allows to relate formal properties of a module to a testbench thereby generating VHDL assertions. But these VHDL assertions do not represent a complete testbench, instead the user needs to create a testbench where these assertions can be used. This is not the case for our approach where our generated scenarios are directly run in a testbench.

8.3.2 Finding Good Test Cases

Finding good scenarios can be considered equivalent to finding good test cases in software development and creating good counterexamples for debugging purposes in hardware development.

The foundation of generating good scenarios in this work are counterexamples, so generating good counterexamples is of special interest to this work. Reference [17] presents heuristics for good counterexamples. The authors propose two heuristics: the first heuristic is a maximum distance heuristic based on three different conditions. The second heuristic builds upon Binary

Decision Diagrams to ensure that no identical counterexamples are chosen. The authors of Ref. [21] propose four algorithms for computing small counterexamples that approximate the shortest case that are based on Dijkstra's algorithm for maximal strongly connected components [13]. But the only applied criterion for a good counterexample is the size. This is not the case for our work, where we also consider the understanding of the generated scenario that will be used for debugging. The work by Ref. [20] deals with this problem by generating different variations of a counterexample that are then used by various analysis routines to provide the user with e.g., control locations of the error.

9. Conclusions

In this article, we have presented a BDD methodology for hardware design. First, we have introduced the integration of testbenches as well as the specification of timing in BDD. Then, we have presented a two-fold BDD strategy which allows for an elegant switch between testing and formal verification. On the one hand, properties can be generalized from test code which can be formally verified on the hardware design. On the other hand, for failing properties debug scenarios can be generated in natural language. For each contribution we have given the respective algorithm and examples for demonstration. Overall, the presented two-fold BDD methodology brings natural language based agile development to hardware design. Thereby, testing and formal verification – which can be used side by side – drive the implementation.

For future work we want to evaluate the developed methods in strong cooperation with an industrial partner.

Acknowledgments This work was supported in part by the German Research Foundation (DFG) within the Reinhart Koselleck project DR 287/23-1 and by the University of Bremen's graduate school SyDe, funded by the German Excellence Initiative.

References

[1] Accellera: Accellera Property Specification Language Reference Manual, Version 1.1 (2005), available from ⟨http://www.pslsugar.org⟩.
[2] Baumeister, H.: Combining Formal Specifications with Test Driven Development, *XP/Agile Universe*, pp.1–12 (2004).
[3] Baumeister, H., Knapp, A. and Wirsing, M.: Property-driven development, *Proc. 2nd International Conference on Software Engineering and Formal Methods*, *SEFM 2004*, pp.96–102, IEEE (2004).
[4] Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R. and Majumdar, R.: Generating Tests from Counterexamples, *Proc. 26th International Conference on Software Engineering*, *ICSE '04*, pp.326–335, IEEE Computer Society (2004).
[5] Biere, A., Cimatti, A., Clarke, E.M., Strichman, O. and Zhu, Y.: Bounded model checking, *Advances in Computers*, Vol.58, pp.117–148 (2003).
[6] Biere, A., Cimatti, A., Clarke, E.M. and Zhu, Y.: Symbolic Model Checking without BDDs, *Tools and Algorithms for Construction and Analysis of Systems*, pp.193–207, Springer (1999).
[7] Bradley, A.R.: SAT-Based Model Checking without Unrolling, *VMCAI*, Jhala, R. and Schmidt, D.A. (Eds.), Lecture Notes in Computer Science, Vol.6538, pp.70–87, Springer (2011).
[8] Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L. and Hwang, L.-J.: Symbolic model checking: 10^{20} states and beyond, *Proc. 5th Annual IEEE Symposium on Logic in Computer Science*, *LICS '90*, pp.428–439, IEEE (1990).
[9] Clarke, E., Grumberg, O., Jha, S., Lu, Y. and Veith, H.: Counterexample-guided abstraction refinement, *Computer Aided Verification*, pp.154–169, Springer (2000).
[10] Clarke, E.M. and Emerson, E.A.: Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic, *Logic of Programs, Workshop*, pp.52–71, Springer-Verlag (1982).
[11] Diepenbeck, M., Kühne, U., Soeken, M. and Drechsler, R.: Behaviour Driven Development for Tests and Verification, *Tests and Proofs*, Seidl, M. and Tillmann, N. (Eds.), Lecture Notes in Computer Science, Vol.8570, pp.61–77 (2014).
[12] Diepenbeck, M., Soeken, M., Große, D. and Drechsler, R.: Behavior Driven Development for circuit design and verification, *Int'l Workshop on High Level Design Validation and Test Workshop* (*HLDVT*), pp.9–16 (2012).
[13] Dijkstra, E.W.: Finding the Maximum Strong Components in a Directed Graph, *Selected Writings on Computing: A Personal Perspective*, Texts and Monographs in Computer Science, pp.22–30, Springer New York (1982).
[14] Dwyer, M.B., Avrunin, G.S. and Corbett, J.C.: Property Specification Patterns for Finite-state Verification, *Proc. 2nd Workshop on Formal Methods in Software Practice*, *FMSP '98*, pp.7–15, ACM (1998).
[15] Dwyer, M.B., Avrunin, G.S. and Corbett, J.C.: Patterns in Property Specifications for Finite-state Verification, *Proc. 21st International Conference on Software Engineering*, *ICSE '99*, pp.411–420, ACM (1999).
[16] Eisner, C. and Fisman, D.: *A Practical Introduction to PSL* (*Series on Integrated Circuits and Systems*), Springer (2006).
[17] Fey, G. and Dreschler, R.: Finding good counter-examples to aid design verification, *Proc. 1st ACM and IEEE International Conference on Formal Methods and Models for Co-Design*, pp.51–52, IEEE Computer Society (2003).
[18] Flanagan, D. and Matsumoto, Y.: *The Ruby Programming Language*, O'Reilly Media (2008).
[19] Foster, H.D.: Trends in functional verification: A 2014 industry study, *Design Automation Conference*, pp.48:1–48:6 (2015).
[20] Groce, A. and Visser, W.: What Went Wrong: Explaining Counterexamples, *Proc. 10th International Conference on Model Checking Software*, *SPIN '03*, pp.121–136, Springer-Verlag (2003).
[21] Hansen, H. and Geldenhuys, J.: Cheap and Small Counterexamples, *6th IEEE International Conference on Software Engineering and Formal Methods*, *SEFM '08*, pp.53–62 (2008).
[22] Henzinger, T.A., Jhala, R., Majumdar, R. and Sanvido, M.A.A.: Extreme model checking, *International Symposium on Verification: Theory and Practice*, LNCS, Vol.2772, pp.332–358, Springer (2003).
[23] Jahanpour, M. and Mohamed, O.: Automatic generation of model checking properties and constraints from production based specification, *Midwest Symposium on Circuits and Systems*, pp.435–438 (2004).
[24] Johnson, N. and Foster, H.: An Agile Evolution in SoC, Presentation at DAC (2015), available from ⟨https://verificationacademy.com/resource/43899⟩.
[25] Johnson, N. and Morris, B.: A Giant Baby Step Forward: Agile Techniques for Hardware Design, *SNUG* (2009).
[26] Martin, G., Bailey, B. and Piziali, A.: *ESL Design and Verification: A Prescription for Electronic System Level Methodology*, Morgan Kaufmann Publishers Inc. (2007).
[27] Morris, B. and Saxe, R.: SVUnit: Bringing Test Driven Design Into Functional Verification, *SNUG* (2009).
[28] Nummenmaa, T., Tiensuu, A., Berki, E., Mikkonen, T., Kuittinen, J. and Kultima, A.: Supporting Agile Development by Facilitating Natural User Interaction with Executable Formal Specifications, *SIGSOFT Softw. Eng. Notes*, Vol.36, No.4, pp.1–10 (2011).
[29] Oberkönig, M., Schickel, M. and Eveking, H.: Improving Testbench Evaluation Using Normalized Formal Properties, *International Conference on Verification and Evaluation of Computer and Communication Systems*, *VECoS '09*, pp.73–83, British Computer Society (2009).
[30] Pnueli, A.: The Temporal Logic of Programs, *FOCS*, pp.46–57, IEEE Computer Society (1977).
[31] Ravi, K. and Somenzi, F.: Minimal Assignments for Bounded Model Checking, *Tools and Algorithms for the Construction and Analysis of Systems*, Jensen, K. and Podelski, A. (Eds.), Lecture Notes in Computer Science, Vol.2988, pp.31–45, Springer Berlin Heidelberg (2004).
[32] Rose, S., Wynne, M. and Hellesøy, A.: *The Cucumber for Java Book: Behaviour-Driven Development for Testers and Developers*, The Pragmatic Bookshelf (2015).
[33] Schlipf, T., Büchner, T., Fritz, R. and Helms, M.: An easy approach to formal verification, *Proc. 10th Annual IEEE International ASIC Conference and Exhibit*, pp.120–124 (1997).
[34] Sheeran, M., Singh, S. and Stålmarck, G.: Checking Safety Properties Using Induction and a SAT-Solver, *FMCAD*, Hunt, Jr., W.A. and Johnson, S.D. (Eds.), Lecture Notes in Computer Science, Vol.1954, pp.108–125, Springer (2000).
[35] Suhaib, S.M., Mathaikutty, D.A., Shukla, S.K. and Berner, D.: Extreme formal modeling (XFM) for hardware models, *5th International*

Workshop on Microprocessor Test and Verification, *MTV04*, pp.30–35 (2004).
[36] Suhaib, S.M., Mathaikutty, D.A., Shukla, S.K. and Berner, D.: XFM: An incremental methodology for developing formal models, *ACM Trans. Des. Autom. Electron. Syst.*, Vol.10, No.4, pp.589–609 (2005).
[37] Sülflow, A., Kühne, U., Fey, G., Große, D. and Drechsler, R.: WoLFram – A Word Level Framework for Formal Verification, *Proc. IEEE/IFIP International Symposium on Rapid System Prototyping*, *RSP '09*, pp.11–17, IEEE (online), DOI: 10.1109/RSP.2009.21 (2009).
[38] Vardi, M.Y.: An Automata-Theoretic Approach to Linear Temporal Logic, *Logics for Concurrency - Structure versus Automata* (*Proc. 8th Banff Higher Order Workshop*), pp.238–266 (1995).
[39] Visser, W., Pasareanu, C.S. and Khurshid, S.: Test input generation with Java PathFinder, *ACM/SIGSOFT International Symposium on Software Testing and Analysis*, *ISSTA*, Avrunin, G.S. and Rothermel, G. (Eds.), pp.97–107, ACM (2004).
[40] Wynne, M. and Hellesøy, A.: *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*, The Pragmatic Bookshelf (2012).

Melanie Diepenbeck finished her diploma in computer science in 2011. Her research focuses on agile development methods and formal verification for hardware design. During her Ph.D. studies she was a student of the graduate school System Design (SyDe). She received her Dr.-Ing. degree from the University of Bremen in 2015. Since 2015 she is working in the automotive industry developing software tools for the GETRAG B.V. & Co. KG.

Ulrich Kühne received his Dr.-Ing. degree in computer science from the University of Bremen, Germany, in 2009. He worked as a postdoctoral researcher in the Laboratoire Spécification et Vérification at ENS de Cachan, France, from 2010 to 2011. He then joined the group of Computer Architecture in Bremen as postdoctoral researcher and coordinator of the Graduate School System Design (SyDe) from 2011 to 2015. Since 2016, he is assistant professor in the Complex Digital Electronic Systems group at Télécom ParisTech, France. His research interests include hardware security, formal verification, and hybrid and timed system analysis. He served as program chair of the PROOFS workshop on security proofs for embedded systems 2017. He is guest editor of the Journal of Cryptographic Engineering (JCEN).

Mathias Soeken received his Dr.-Ing. degree in computer science from the University of Bremen, Germany, in 2013. He works as a researcher at the Integrated Systems Laboratory at EPFL, Lausanne, Switzerland. From 2009 to 2015 he worked at the University of Bremen, Germany. Since 2014, he is a regularly visiting post doc at UC Berkeley, CA, USA. His research interests are logic synthesis, quantum computing, reversible logic, reverse engineering, and formal verification. He published more than 100 papers in peer-reviewed journals and conferences and serves in program committees of numerous conferences including DAC, ICCAD, DATE, and FMCAD. He is member of the IEEE and ACM.

Daniel Große received his Dr.-Ing. degree in computer science from the University of Bremen, Germany, in 2008. He remained as postdoctoral researcher in the group of Computer Architecture in Bremen. In 2010 he was a substitute professor for computer architecture at Albert-Ludwigs University, Freiburg, Germany. From 2013 to 2014 he was CEO of the EDA start-up solvertec focusing on automated debugging techniques. Since 2015 he is a senior researcher at the University of Bremen and the German Research Center for Artificial Intelligence (DFKI) Bremen and also the scientific coordinator of the graduate school System Design (SyDe), funded within the German Excellence Initiative. His research interests include verification, high-level languages, virtual prototyping, debugging and synthesis. In these areas he published more than 100 papers in peer-reviewed journals and conferences and served in program committees of numerous conferences like e.g., DAC, ICCAD, DATE, CODES+ISSS, FDL, MEMOCODE.

Rolf Drechsler received his Diploma and Dr. Phil. Nat. degrees in computer science from J.W. Goethe University Frankfurt am Main, Frankfurt am Main, Germany, in 1992 and 1995, respectively. He was with the Institute of Computer Science, Albert-Ludwigs University, Freiburg im Breisgau, Germany, from 1995 to 2000, and with the Corporate Technology Department, Siemens AG, Munich, Germany, from 2000 to 2001. Since October 2001, he has been with the University of Bremen, Bremen, Germany, where he is currently a Full Professor and the Head of the Group for Computer Architecture, Institute of Computer Science. In 2011, he additionally became the Director of the Cyber-Physical Systems group at the German Research Center for Artificial Intelligence (DFKI) in Bremen. His current research interests include the development and design of data structures and algorithms with a focus on circuit and system design. He is an IEEE Fellow. He was a member of Program Committees of numerous conferences including e.g., DAC, ICCAD, DATE, ASP-DAC, FDL, MEMOCODE, FMCAD, Symposiums Chair ISMVL 1999 and 2014, Symposium Chair ETS 2018, and the Topic Chair for "Formal Verification" DATE 2004, DATE 2005, DAC 2010, as well as DAC 2011. He received best paper awards at the Haifa Verification Conference (HVC) in 2006, the Forum on specification & Design Languages (FDL) in 2007 and 2010, the IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS) in 2010 and the IEEE/ACM International Conference on Computer-Aided Design (ICCAD) in 2013. He is an Associate Editor of IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on Very Large Scale Integration Systems, Design Automation for Embedded Systems, IET Cyber-Physical Systems: Theory & Applications, International Journal on Multiple-Valued Logic and Soft Computing, and ACM Journal on Emerging Technologies in Computing Systems.

(Invited by Editor-in-Chief: *Nozomu Togawa*)