Behaviour Driven Development for Tests and Verification

Melanie Diepenbeck[1], Ulrich Kühne[1], Mathias Soeken[1,2], and Rolf Drechsler[1,2]

[1] Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
[2] Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{diepenbeck,ulrichk,msoeken,drechsle}@informatik.uni-bremen.de

Abstract. The design of hardware systems is a challenging and error-prone task, where a signifcant portion of the effort is spent for testing and verification. Usually testing and verification are applied as a post-process to the implementation. Meanwhile, for the development of software, test-first approaches such as *test driven development* (TDD) have become increasingly important. In this paper, we propose a new design flow based on *behaviour driven development* (BDD), an extension of TDD, where acceptance tests written in natural language drive the implementation. We extend this idea by allowing the specification of properties in natural language and use them as a starting point in the design flow. The flow also includes an automatic generalisation of test cases to properties that are used for formal verification. In this way, testing and formal verification are combined in a seamless manner, while keeping the requirements — from which both tests and formal properties are derived — in a single consistent document. The approach has been implemented and evaluated on several examples to demonstrate the advantages of the proposed flow.

1 Introduction

In the design of hardware and software systems, testing and verification are often more labour and cost intensive than the implementation itself. The higher the quality standards — up to safety critical systems in cars, avionics, or medical equipment — the more time needs to be spent in writing good test benches or formal properties. In traditional hardware design flows, verification is often done at post-design time. This practice can result in long design cycles, since serious bugs discovered at this late stage might lead to major design changes or modifications of the specification. This is why it is desirable to start the validation as early as possible.

In the software domain, agile techniques have become quite popular, as a means to shorten the design cycles and achieve a more flexible flow, where changes can be integrated quickly. In *test driven design* (TDD), the tests are written first [1], which forces the designer to think about the requirements and interfaces before getting started with coding. *Behaviour driven development* (BDD) is an extension of TDD, where the tests are written in natural language [2]. In BDD, textual scenarios, which can easily be derived from the

requirements, provide a valuable link between the specification and the implementation. During the design process, the scenarios are ported step by step to executable tests. Writing tests and implementing the required code is interleaved, resulting in short design cycles. There have been some attempts to make use of agile techniques in the hardware domain [3], also with a focus on formal models and verification techniques [4, 5].

In the context of BDD, the natural language scenarios are usually used to describe acceptance tests, i.e. scenarios that test whether certain features are implemented according to the requirements. But, when applied to safety critical hardware designs, just testing is not enough. Since it is infeasible to cover the whole input and state space even of smaller hardware blocks by mere simulation, there remains a risk that subtle bugs will be missed. This is where formal methods come into play. Using automatic or semi-automatic proof techniques, high confidence can be reached in the correct functionality. In particular, SAT-based model checking techniques like [6–8] have been successfully applied to industrial scale hardware designs. However, their application is difficult and requires writing properties in dedicated languages such as the *property specification language* (PSL, [9]).

In this paper, we present for the first time a hardware design flow that completes the BDD method by complementing tests with formal verification techniques in a flexible and agile way. The methodology builds on the popular BDD tool *cucumber* [2]. We enhance the existing test driven flow by integrating formal verification, while keeping the natural language requirements in a single consistent document. As a first step to improve the design quality, test cases can be generalised automatically to PSL properties. This enables the use of model checking tools with no additional effort for the user. However, not all test cases can be generalised in this way. We add further flexibility by allowing to write requirements dedicated to formal verification only, that will be translated directly to PSL. This allows the use of more powerful constructs, which cannot easily be described by single test cases.

Overall, the contributions of this work are the following:

- Strengthening BDD for hardware design by automated test generalisation
- A seamless integration of tests and formal properties in a single human readable document
- The implementation within the popular BDD tool cucumber
- The experimental evaluation on several examples

The paper is structured as follows. First, the used property specification language and the basics of BDD are introduced in Sect. 2. The proposed BDD flow for hardware design and verification are presented in Sect. 3. Section 4 discusses advantages and limitations of our approach. Related work can be found in Sect. 5. The paper is concluded in Sect. 6.

2 Background

2.1 Property Specification Language

PSL has been adapted as IEEE standard 1850 in 2005 [9]. It is supported by many verification tools, both dynamic (simulative) and formal. PSL comes in different *flavors* for the hardware description languages VHDL, Verilog, and SystemVerilog, as well as for SystemC, a C++ library for hardware and system design.

The language is organised in *layers*, starting at the bottom with the Boolean layer, which consists of expressions from one of the flavor languages. On top of this, timing can be added in the temporal layer. Basically, PSL is a superset of *linear time logic* (LTL, [10]). Besides the standard operators such as `always`, `until`, and `next`[3], a convenient way to describe computations in PSL are *sequentially extended regular expressions* (SEREs). Like ordinary regular expressions, they allow for pattern matching and provide an easy way to express repetitions and concatenation, but focusing on temporal aspects. Finally, at the verification layer, directives are given to the verification tool, what to do with the stated properties. Here, we will only introduce a subset of this very rich standard. A good introduction to PSL for hardware verification can be found in [11].

Example 1. Consider the following simple PSL property:

```
property reset = always {rst; !rst} |-> pc == 0;
```

The operator `always` indicates that the following expression should be considered an invariant, which must hold at *every* single cycle in *any* computation. The invariant given by property `reset` is formed by the *overlapping suffix implication* operator '`|->`'. On the left hand side of the suffix implication, a SERE is formed, consisting of two cycles, where the signal `rst` is high in the first and low in the second cycle. The suffix implication has the following meaning: if the left hand side sequence occurs, then the right hand side must hold simultaneously to the last cycle of the sequence. Overall, the property `reset` states that after asserting and releasing signal `rst`, the signal `pc` must be set to zero. A slightly more complex property is stated below:

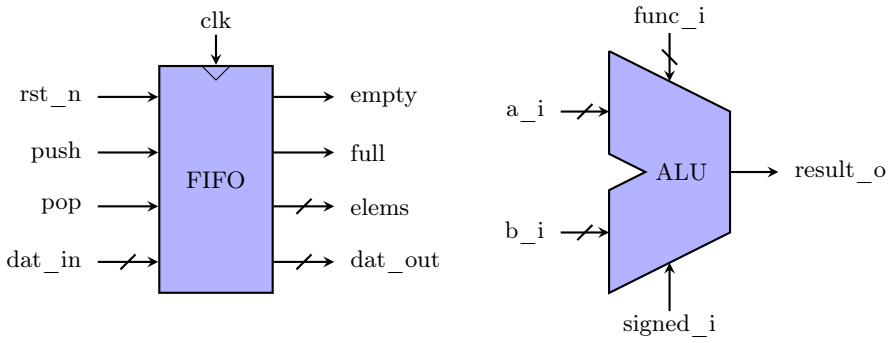
```
property release_req = always {{req; ack} | {req; rty}} |=> !req;
```

Here, the operator '`|=>`' is called *non-overlapping suffix implication*. The property holds if the right hand side holds in the cycle directly *after* the last cycle of the sequence on the left hand side. The left hand side SERE is a composition of two sequences, combined with the *non-length matching or* '`|`'. The left hand side matches if any of the two given sequences match. Overall, the property says that `req` should be released after it has been asserted and after either `ack` or `rty` has occurred.

[3] according, respectively, to `G`, `U` and `X` in LTL

Besides the operators in the above examples, we will use the *non-length matching and*, expressed by a single '`&`', which combines two sequences analogously to the non-length matching or. The built-in functions `next`(φ) and `prev`(φ) can be used to retrieve the value of an expression φ in the next or the previous cycle, respectively. The built-in function `stable`(φ) is a shortcut for the expression $\varphi == $ `prev`(φ).

2.2 Running Examples



(a) First-in-first-out queue (b) Arithmetic-logic unit

Fig. 1. Block diagrams of example circuits

As a running example, a *first-in-first-out queue* (FIFO) and an *arithmetic-logic unit* (ALU) will be used. Figure 1 shows block diagrams of the designs under test. The FIFO is a synchronous circuit, driven by the clock signal `clk`. The single bit outputs `empty` and `full` give information on the fill status of the FIFO, while `elems` shows the exact number of elements currently in the queue. The oldest data element can be read from output `dat_out`. By asserting the input `rst_n` (active low), the FIFO is cleared. Elements are added and removed using the signals `push` and `pop`, respectively, where `dat_in` is used to present the data to be added to the queue. The actual design under test has a capacity of four elements.

The ALU in Fig. 1(b) computes 2-input logic and arithmetic functions. The type of function is selected via the five bit input `func_i`. The ALU implements 17 different functions, among them addition, shifting, multiplication and comparisons like equals or less. The single bit input `signed_i` indicates whether both the 32 bit data inputs are to be treated as signed or unsigned integers. The ALU is part of an open source hardware project.[4]

[4] http://opencores.org/project,m1_core

Scenario Outline: *Pushing*
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

```
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. 2. BDD scenario with step definitions

2.3 Behaviour Driven Development

BDD extends the idea of TDD with natural language written user stories or acceptance tests, which are called *scenarios*. These are grouped by means of *features* and each scenario is described as a sequence of *sentences*.

Example 2. Figure 2(a) shows an example scenario that describes how data is written (push operation) into 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.

In order to have a nicely readable text, the BDD flow suggest to use the keywords *Given*, *When*, and *Then*, that refer to test code containing assumptions, conditions, and assertions, respectively. Note that these keywords have no further semantic meaning in the BDD tool. The keyword *And* can be used to avoid repetition and one can also use * as a generic keyword to introduce a sentence. In fact, the keyword does not even have to match the keyword used in the step definitions. Consequently, *Then* sentences can e.g. also be used as *When* sentences.

To automatically execute a sentence in a scenario, one has to provide a *step definition* which is a 3-tuple consisting of a keyword, a regular expression, and test code. The BDD tool then essentially works as follows:

1. For each sentence in a scenario it is checked whether it is matched by a regular expression of a step definition. Step definitions are ordered and the first matching step definition is taken.
2. If necessary, values are extracted from the sentence using capture groups in the regular expression. Then, the test code is executed.

Since regular expressions are used in order to match a sentence to a step definition, there is no restriction on the natural language that is used to describe the scenarios (An exception is the approach that is presented in [12] and uses natural language processing techniques to extract structural information to

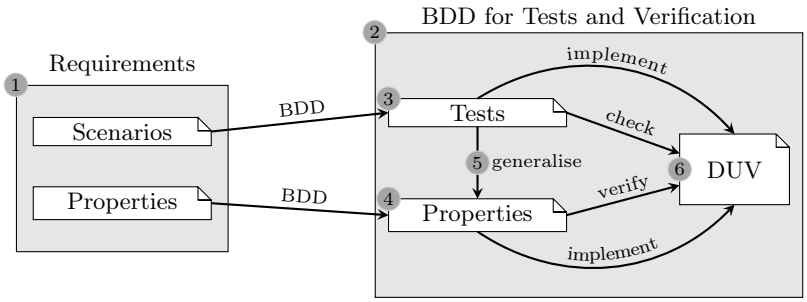


Fig. 3. Improved BDD flow

create a formal model from a set of natural language scenarios). Optionally, it is possible to add special test code that is executed before and after each scenario. Every scenario is executed separately within its own test bench environment.

Example 2 (continued). For each sentence in the scenario the designer creates a step definition, as listed in Fig. 2(b). Since no implementation is available at this point, the designer only decides on the input and output signals of the FIFO module in the step definitions.

A predefined sentence "**And** I wait t cycles" handles timing where test code in succeeding sentences takes place t cycles after the sentence before this timing sentence.

This scenario can now be used for testing, however, step definitions cannot be run directly. Instead, they require a test bench that encloses the test code of all sentences of a scenario. This test bench can either be written by the designer or generated automatically from the design information given by the designed module.

In our examples we mainly use *scenario outlines* instead of scenarios. A scenario outline is a parameterised scenario which is enriched with an *examples table* that allows the specification of several test assignments given by each row in the table. The variables of a scenario outline, denoted e.g. with `<a>`, enable property generalisation, which will be shown in Sect. 3.1.

3 BDD for Tests and Verification

Based on the BDD flow that has been described in the previous section, the idea for the improved flow is introduced in this section. Our proposed flow is driven by tests and properties.

The first thing that is usually done in a BDD based approach is writing test cases in terms of scenarios to drive the implementation. This is a first step in verifying the *design under verification* (DUV), and helps in achieving a good design quality. Nevertheless, this is usually insufficient to completely verify the

design, since the input and state space of non-trivial designs can hardly be covered by test runs. As a first improvement, it is possible to automatically *generalise* test cases to formal properties, thereby covering more potential bugs. However, some requirements cannot be easily stated as test cases. This holds especially for global properties like in the following example.

Example 3. Consider the FIFO example of Sect. 2.2 which has a capacity of four elements. In order to check this requirement of the FIFO, the designer needs to check two scenarios; (1) a scenario which checks if it is possible to insert at least four elements and (2) a scenario that checks that a fifth element cannot be inserted into the queue. Even then, possible bugs — like an underflow when popping from an empty queue — could be missed, which would violate this requirement.

Therefore, we propose to drive the implementation by both tests and properties, as shown on the left hand side of Fig. 3 (marked by `1`), where requirements are described by scenarios and properties. For each sentence of each scenario and property, step definitions are defined to express the desired behaviour of a sentence using test code or PSL expressions. This code is then used to guide the implementation of the design (`2`). The properties and tests are used in all stages of the BDD approach to implement, test, and verify the DUV (`3,4,6`). By generalising tests to properties (`5`), the verification is strengthened. The generalisation even allows to reuse parts of a test scenario in a property.

In the remainder of this section it is described how test cases can be generalised, but also limitations are drawn. It is then shown how to overcome these limitations by writing properties in addition to scenarios to cover more requirements of the DUV.

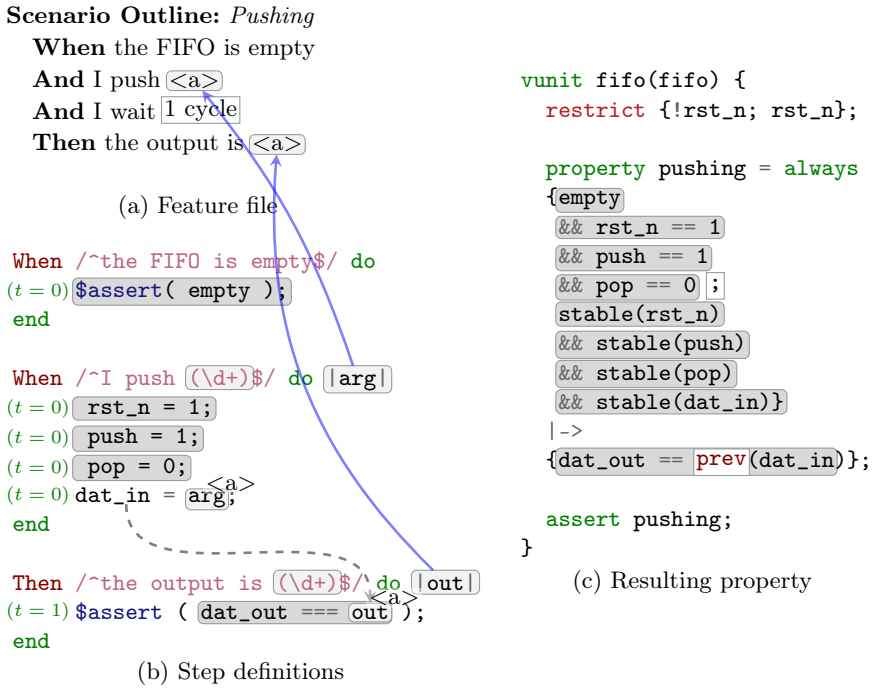
3.1 From Tests to Generalised Properties

As described in Sect. 2.3, acceptance tests are used to create a circuit design using Verilog. The tests are created in a BDD manner, as shown in Fig. 3 and can be used to check the DUV.

Acceptance tests as illustrated in Example 2 usually consider only few selected test input data and never cover a scenario exhaustively. Such scenarios can be generalised in terms of a PSL property which covers 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. While the antecedent of the implication property is filled with the step definition code of all *When*-steps of a scenario, 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. A property is generated as follows.

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

P1. [Sorting step definitions.] The step definition code of every step of the scenario is mapped to the appropriate part of the property.



(a) Feature file

(b) Step definitions

(c) Resulting property

Fig. 4. From a scenario to a generalised property

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 needs to be extracted. Every statement of the step definition code 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.

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.

Figure 4 illustrates the briefly sketched algorithm P. All statements in Fig. 4(b) that have a grey background will be inserted into the appropriate part of the property, depending on whether the step is a *When* or *Then*-step, as described in step P1. The last statement of the second step definition is left out, because it only assigns test input data.

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. 4(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.

For timing consideration each statement is annotated with a current time t. The timing of the statements can be seen in Fig. 4(b) on the left side of each statement. The predefined Timing sentence "**And** I wait t cycles" increments the current time step.

Before the complete property can be assembled, it is necessary to consider the test semantics first. While testing, the input signals are assigned imperatively. A new time step does not change the value of the signals unless explicitly specified. Since the imperative semantics of test code is not implicitly considered in properties for verification, these test semantics need to be ensured explicitly. For signals that do not change, a `stable`-expression is inserted to ensure the value of the signal stays the same. These statements can be seen in the resulting property in Fig. 4(c).

In the last step, the property is assembled using all gathered informations. The timing informations gathered in step P3 is used to assemble the SERE expression for the antecedent and consequent. This can be seen in Fig. 4(c) in the antecedent of property `pushing` where a `';'` is added between the expressions of the first and the second time step. In the antecedent the second time step consists of all `stable`-assignments of all unchanged signals.

SEREs provide a concise and flexible way to express complex properties, in particular when compared to the initial approach in [13], which only allowed the translation of rather simple test cases. For instance, in this style, the generalisation of an if-then-else statement can be represented as follows:

```
{{{condition} & {if_assigns}} | {{!condition} & {else_assigns}}}
```

3.2 Limitation of Test Generalisation into Properties

The method to generalise properties as discussed in the previous section cannot be applied to all scenarios. Limitations of the approach are listed in the section, and two types of test cases that lead to invalid properties are illustrated.

Example 4. Consider the exemplary generalisation of the scenario for an addition operation of the ALU in Fig. 5. Figure 5(a) shows a valid scenario for this requirement. The step definitions for this scenario are given in Fig. 5(b). The first two step definitions set the inputs to the values from the examples table. When this test is generalised the design inputs `a_i` and `b_i` are detected as inputs

Scenario Outline: *Adding*
When I set the 1st operand to <a>
And I set the 2nd operand to <b>
And I want to add these
Then the output should be <c>
Examples:

a	b	c
10	15	25
20	15	35
-20	15	-5

(a) Feature file

```
1 When /^I set the 1st operand to
       (\d+'b\d+)$/ do |arg|
    a_i = arg;
  end

2 And /^I set the 2nd operand to
       (\d+'b\d+)$/ do |arg|
    b_i = arg;
  end

3 And /^I want to add these$/ do
    signed_i = 1;
    func_i = 4;
  end

4 Then /^the result should be
       (-?\d+)$/ do |arg|
    $assert( result_o === arg );
  end
```

(b) Step definitions

```
property addition_signed = always
  {signed_i==1 && func_i==4} |-> {result_o == ? };
```

(c) Resulting property

Fig. 5. Missing input-output-relationship

that can have arbitrary values. The fourth step definition compares the result of the design output with the given value for <c> in the table. But the assertion of the design output `result_o` cannot be generalised since <c> cannot be mapped to anything in the design. That is because no input-output-relationship is known. This is shown in the property of Fig. 5(c) where `arg` is replaced by a question mark. Although <c> is the addition of <a> and <b>, the relation is never explicitly stated. Therefore this scenario can not be generalised and would be skipped.

The designer could make this scenario generalisable when she restates the last step to "**Then** the output should be the addition of <a> and <b>" and creates a complying step definition for it that explains the relation.

```
Then /^the result should be the sum of (-?\d+) and (-?\d+)$/
do |arg1, arg2|
    $assert( result_o === arg1 + arg2 );
end
```

There might also be test cases where generalisation does not yield a useful property. This is often the case when the relationship of the design *input* signals is relevant.

Example 5. Consider the relation operator '<' of the ALU module. A typical scenario for an acceptance reads as follows:

Scenario Outline: *less than*
When I set the first operand to <a>
And I set the second operand to <b>
And I use the less-than operator
Then the output should be true
Examples:

a	b
10	15
400	512
-40	15

(1)

The property that is being generalised from this scenario and the corresponding step definitions is:

```
property less_than = always {signed_i==1 && func_i==13} |-> {result_o==1};
```

Although the property can be generalised it will fail when verifying it against the implementation. The relationship of the inputs `a_i` and `b_i` cannot be derived from the test case. A specific counter example for this property is that the design input `a_i` is set to 7, while `b_i` is set to 0. Therefore `a_i` (respectively `<a>`) is greater than `b_i` (respectively `<b>`). Again, this relationship has only been stated implicitly in the examples table. This is why the property will fail when considering arbitrary values of `<a>` and `<b>`.

In this case, the user needs to make the relationship explicit by adding it as an assumption. This can be done using *Given*-sentences. In the above example, the designer can fix the property by adding the sentence "**Given** `<a>` is less than `<b>`" to the beginning of the scenario. This *Given*-sentence is then translated to a global assumption, using an `assume` directive in PSL:

```
property is_less_than = always {a_i < b_i};
assume is_less_than;
```

The refactoring of scenarios can lead to a better code inspection and therefore improves the design understanding. In general, it is easy to rephrase the scenarios in order to apply property generalisation. But it may also be desirable to treat a scenario as a normal test case. We offer this possibility by annotating a scenario with a tag indicating it must not be generalised.

3.3 Specifying Properties

As motivated in the previous section, the property generalisation approach has its limitations. However, when disabling property generalisation for certain scenarios, an exhaustive consideration of the input space is no longer guaranteed. As an alternative, we extended the features to contain standalone properties next to scenarios. They work like scenarios with the difference that step definitions do not contain test code but PSL code to build the property. Consequently, properties are checked using an automatic formal verification tool and not executed

Property: *Incrementing*
When the FIFO is not empty
And I push an element
And I wait 1 cycle
Then the number of elements has increased

(a) Property

```
When /^the FIFO is
       not empty$/ do
Verilog::add_antecedent do
  !empty
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. 6. Implication property

as part of a test bench. On the level of the natural language requirements, there is no difference between plain properties and test cases.

A natural language property can be specified in two different ways: (1) as an implication property similar to the generated properties given by the *Given-When-Then*-structure or (2) as an invariant property without using the provided structure.

Specifying natural language implication properties. In the following we illustrate how to specify natural language implication properties. The structuring of this type of properties is very similar to the specification of scenarios.

Example 6. Figure 6(a) shows how to write an implication property that states that the number of elements of a FIFO is increased whenever an element is pushed. The property looks very similar to a scenario which is used for testing, but instead of writing test code, the desired behaviour is expressed with PSL code. The step definitions in PSL are given in Fig. 6(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.

Specifying natural language invariant properties. When an implication structure is not needed to express a property, the usual *When* and *Then* keyword may be superfluous.

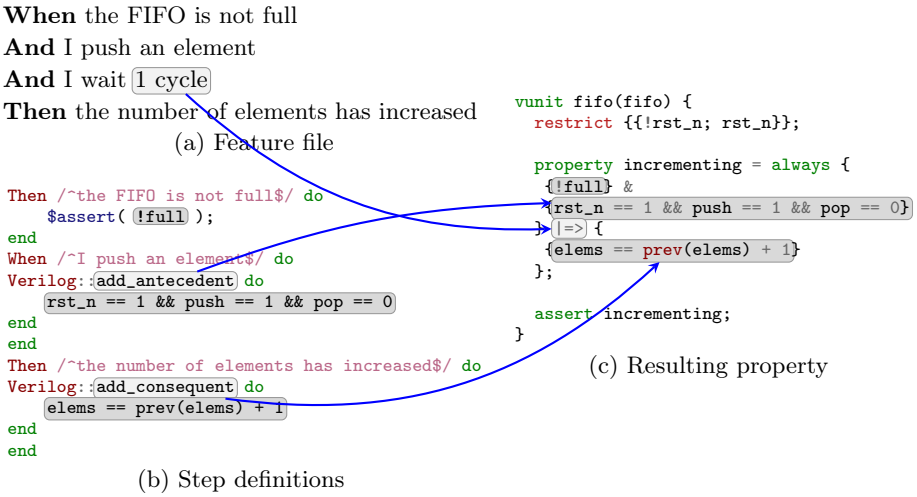


Fig. 7. Implementation

Example 7. As an example for a simple invariant, this property fixes 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. (2)

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_antecedent do
  elems <= 4
end
end
```

In this case it may seem that a comment to a PSL property would also suffice to describe the property, but the difference is that the natural language description also serves as a specification. Therefore the invariant property is an important part of the feature description of the design.

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. Similar to the property generalisation, the PSL code from the antecedent parts and the consequent parts of the step definitions are mapped to the antecedent and the consequent of the resulting property, as can be seen in Fig. 7.

The PSL code of each step is joined for the antecedent and consequent block, respectively. If the property code of the antecedent (respectively consequent)

Table 1. Examples used to evaluate the new flow

Design	#Scenarios	#Properties	#Gen. Properties
FIFO	7	4	5
m1_alu	22	19	8
counter	1	4	1
hamming	6	5	2

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 as it is done in the generalisation.

In the property in Fig. 7, the timing is explicitly stated by one of the steps that 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 cylce after the last cycle of the antecedent.

The first step definition in the example in Fig. 7 is particularly interesting, since it is written in Verilog test code. This example shows that it is additionally possible to reuse sentences that are used in acceptance tests. To add this test code to the resulting property it is necessary to generalise the statement. The step definition code is assumed to belong to the antecedent block due to the *When* keyword and inserted into the antecedent as it is done in the property generalisation.

Because *assume*-blocks describe global restrictions, an independent property is assembled for every *assume*-block given in a property, which is then assumed in the generated PSL code. This is analogous to the transcription of the *Given*-sentences. This is not shown in the example in Fig. 7.

4 Discussion

In this section we discuss the new flow which has been implemented on top of the *cucumber* tool in Ruby. We use WoLFram [14] as the underlying model checker. Our approach was applied to several examples which are listed in Table 1. The table states the number of specified scenarios and properties that have been used to drive the implementation in the first two columns. The third column states the number of properties that could be generalised from the specified scenarios. Every functionality of each example is verified using properties that have either been written or been generalised from the specified scenarios.

In the following we discuss the advantages that arise with the new proposed flow by referring to some of the examples on which we applied our approach. The first point to be noticed is the previously explained limitation of the property generalisation which is described in Sect. 3.2. In the proposed approach of [13] the generalised properties were the only possibility to formally verify the design. Therefore, it was sometimes necessary to rephrase the scenario or add a new

Scenario Outline: *and operation*
When I set the first input to <a>
And I set the second input to <b>
And I use the AND operation
Then the result is the logical AND of <a> and <b>
Examples:

a	b
2'b11	2'b00
2'b01	2'b10
2'b00	2'b00
2'b11	2'b01

Property *and operation*
When I want to and two operands
Then the result is the logical AND

Fig. 8. Scenarios vs. Properties

step in order to create a valid property. While this is generally easy, it is rather uncommon when defining tests. In this case, the direct mapping from a scenario to a property is a convenient alternative.

In Fig. 8 the specification of scenarios and properties for the logical AND operation of the ALU example can be compared. It is apparent that stating the requirements in a scenario is more long-winded than the specification of the requirement as a property. This shows an advantage of this approach since requirements can be described more concisely which improves the discussion with stakeholders. This could be observed while specifying tests for the 17 functions of the ALU.

Also some requirements cannot be easily stated just using scenarios. Consider the invariant from the previous section that stated that the FIFO can only contain at most 4 entries.

Property: *Invariant*
***** the number of elements in the FIFO is at most 4

In order state this invariant as acceptance tests, the designer would need to write several scenarios. At least one that states that the FIFO *can* contain 4 elements and another one that states that the FIFO will not take a fifth element after four elements have been inserted. This is very laborious and not very concise. This observation has an important implication on the original idea of BDD; defining properties as "acceptance tests" completes the aspect of behaviour driven development.

However, it is not just that scenarios and properties can be defined in the same document and then tests and verification is used separately. It is even possible to use steps from scenarios that were only designed for testing in the specification of properties. The significance of this became very evident while applying the approach on the examples of Table 1. While implementing the FIFO one third of the step definitions in properties were reused from test scenarios.

Even complete properties were defined using only steps from scenarios, but instead of resulting in executable test code creating a valid provable property.

5 Related Work

A first step in this direction has been presented in [13]. But this approach is one-sided since it only supports a test-based approach that generalises properties. Our new approach additionally supports a property-based BDD which goes hand in hand with the previously presented test-based BDD approach. It is also enhanced by a more advanced property generalisation that supports a more complex test bench design. Both tests and verification are the main driver for the implemented design.

Baumeister proposed an approach of a generalisation of tests to a formal specification in [15]. His work considers Java as target language where the specification is checked using generated JML. The drawback is that the approach does not facilitate an automatic generalisation of tests. In [16] a property driven development approach is presented 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 generalising test cases. But this approach is not implemented.

Agile techniques for hardware design are heavily discussed in several blog post [17]. One of the most promising approaches that was presented is SVUnit [3], which is a unit test framework created for SystemVerilog that enables TDD for circuit design.

The combination of formal techniques and agile design has been considered in [4]. There, Henzinger et al. propose a paradigm called "extreme model checking", where a model checker is used in an incremental fashion during the development of software programs. Another approach is called "extreme formal modeling" [18]. 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 [5].

6 Conclusions

We proposed a new BDD based flow that combines testing and verification in a seamless manner using natural language tests and properties as starting point for the design. For this purpose we introduced a new element for defining properties in natural language and supported the assembling of PSL code to valid properties that can be used for verification. This approach helps designers to write properties by starting from natural language. Our new flow supports the idea of *completeness driven development* (CDD) [19] by also defining properties as a starting point.

Further research will explore how to generate tests from properties written in BDD style. This can be useful to speed up regression tests during the design, when the formal verification of global properties would take too long. Future

work will extend the specification language for properties to make scenarios and properties more expressive and to help the designer in writing properties more easily, especially if he or she is not a verification engineer.

Acknowledgments.

This work was supported by the German Research Foundation (DFG) within the Reinhart Koselleck project DR 287/23-1 and by the Graduate School SyDe, funded by the German Excellence Initiative within the University of Bremen's institutional strategy.

References

1. Beck, K.: Test Driven Development. By Example. Addison-Wesley Longman, Amsterdam (November 2003)
2. Wynne, M., Hellesøy, A.: The Cucumber Book: Behaviour-Driven Development for Testers and Developers. The Pragmatic Bookshelf (January 2012)
3. Morris, B., Saxe, R.: svunit: Bringing Test Driven Design Into Functional Verification. In: SNUG. (2009)
4. Henzinger, T.A., Jhala, R., Majumdar, R., Sanvido, M.A.A.: Extreme model checking. In: International Symposium on Verification: Theory and Practice. Volume 2772 of LNCS., Springer (2003) 332–358
5. Suhaib, S., Mathaikutty, D., Shukla, S., Berner, D.: Extreme formal modeling (XFM) for hardware models. In: Fifth International Workshop on Microprocessor Test and Verification MTV04. (2004) 30–35
6. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic Model Checking without BDDs. In: Tools and Algorithms for Construction and Analysis of Systems, Springer (March 1999) 193–207
7. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In Jr., W.A.H., Johnson, S.D., eds.: FMCAD. Volume 1954 of Lecture Notes in Computer Science., Springer (2000) 108–125
8. Bradley, A.R.: SAT-based model checking without unrolling. In Jhala, R., Schmidt, D.A., eds.: VMCAI. Volume 6538 of Lecture Notes in Computer Science., Springer (2011) 70–87
9. Accellera: Accellera property specification language reference manual, version 1.1. http://www.pslsugar.org (2005)
10. Pnueli, A.: The temporal logic of programs. In: FOCS, IEEE Computer Society (1977) 46–57
11. Eisner, C., Fisman, D.: A Practical Introduction to PSL (Series on Integrated Circuits and Systems). Springer, Secaucus, NJ, USA (2006)
12. Soeken, M., Wille, R., Drechsler, R.: Assisted behavior driven development using natural language processing. In: TOOLS. (2012) 269–287
13. Diepenbeck, M., Soeken, M., Grosse, D., Drechsler, R.: Behavior driven development for circuit design and verification. In: Int'l Workshop on High Level Design Validation and Test Workshop (HLDVT). (Nov 2012) 9–16
14. Sülflow, A., Kühne, U., Fey, G., Grosse, D., Drechsler, R.: WoLFram – A word level framework for formal verification. In: Proceedings of the IEEE/IFIP International Symposium on Rapid System Prototyping. RSP '09, IEEE (2009) 11–17

15. Baumeister, H.: Combining formal specifications with test driven development. In: XP/Agile Universe. (2004) 1–12
16. Baumeister, H., Knapp, A., Wirsing, M.: Property-driven development. In: Software Engineering and Formal Methods, 2004. SEFM 2004. Proceedings of the Second International Conference on, IEEE (2004) 96–102
17. Johnson, N., Morris, B.: AgileSoC. http://www.agilesoc.com/ (2012)
18. Suhaib, S.M., Mathaikutty, D.A., Shukla, S.K., Berner, D.: XFM: an incremental methodology for developing formal models. ACM Trans. Des. Autom. Electron. Syst. **10**(4) (2005) 589–609
19. Drechsler, R., Diepenbeck, M., Große, D., Kühne, U., Le, H.M., Seiter, J., Soeken, M., Wille, R.: Completeness-driven development. In: International Conference on Graph Transformation. Lecture Notes in Computer Science, Springer Berlin Heidelberg (2012) 38–50