Behavior Driven Development for Circuit Design and Verification

Melanie Diepenbeck^1 Mathias Soeken^{1,2} Daniel Große^1 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, msoeken, grosse, drechsle}@informatik.uni-bremen.de

***Abstract*—The design of hardware systems is a challenging and erroneous task where about 70% of the effort in designing these systems is spent on verification. In general, testing and verification are usually tasks that are being applied as a post-process to the implementation.**

In this paper, we propose a new design flow based on *Behavior Driven Development* (BDD), an agile technique for the development of software in which acceptance tests written in natural language play a central role and are the starting point in the design flow. We advance the flow such that the specifics that arise when modeling hardware are taken into account. Furthermore, we present a technique that allows for the automatic generalization of test cases to properties that are suitable for formal verification. This allows the designer to apply formal verification techniques based on test cases without specifying properties.

We implemented our approach and evaluated the flow for an illustrative example that successfully demonstrates the advantages of the proposed flow.

I. Introduction

With the advent of computers the problems of bugs in hardware and software arose. Since then the challenge of "getting the bugs out" continues to grow due to the increasing complexity of the systems. Hence, enormous effort has been put into the development of methodologies reducing the bug rates or even ensuring correctness.

Meanwhile for the development of software *Test Driven Development* (TDD) [1] has become an important approach which has its roots in agile programming [2]. Essentially, in TDD testing and writing code is interleaved while the test cases are written before the code. This ensures that each new functionality is tested a priori. In doing so, testing becomes part of the software development process and it is no longer a post-process, which in practice is often omitted due to strict time constraints. The advantages of TDD have been demonstrated in several industrial studies. For example, [3] reported a reduction of the defect density between 40% and 90% for four products. The author of [4] was able to show that the application of TDD reduced the defect rate by about 50% compared to a similar system that was built using an ad-hoc unit testing approach.

Recently, an extension of TDD has been proposed. In *Behavior Driven Development* (BDD) [5] the test cases are written in natural language which enables the discussion with stakeholders since they do not have to read code. The used BDD language terminology focuses on the behavioral aspects rather than on testing: The *Given-When-Then* sentence structure of the tests connects the human concept of cause and effect to the software concept of input/process/output in an intuitive way [6].

In this paper we present a new flow for circuit design and verification based on BDD. Our contribution is twofold: First, we customize BDD for circuits modeled in a *Hardware Description Language* (HDL). Therefore, we have to take the specifics of circuits, in particular timing and testbenches, into account. As a result, circuit components are iteratively created and, based on the underlying BDD methodology, the implemented functionality is tested in a structured way while the tests are connected to natural language specifications. Second, given the specific structure of the acceptance tests we present a technique to automatically generalize these tests. That is, we are able to generate properties from the written test code. These properties are formally verified on the iteratively developed implementation and hence enable to fully exhaust the potential of the test cases. Both contributions are integrated into a new flow enabling the following advantages:

- Circuit design starting from a natural language based specification down to an HDL implementation
- Formal verification without manual specification of complex properties
- Improved verification quality based on automatically generated properties

We have implemented the proposed flow. In the experimental evaluation the hardware implementation of a vending machine is presented. We discuss the design as well as the verification of the system and demonstrate the advantages of our new flow.

The remainder of this paper is structured as follows: Section II gives the background on BDD. Then, in Section III the proposed flow is introduced while Section IV details the implementation. An illustrative example is described in Section V and Section VI discusses related work. Finally, the paper is concluded in Section VII.

II. Behavior Driven Development

TDD is a design flow paradigm in which test cases are provided as starting point and central elements along the whole design process. First, all test cases are specified, however, since no implementation is available, they will all fail initially. Based on the error messages from the failing test cases, the implementation grows incrementally until all test cases pass eventually. Based on this design flow paradigm, BDD has been recently proposed in which the test cases are specified using natural language rather than source code thereby offering a ubiquitous communication mean for both the designers and stakeholders. The natural language ensures a common understanding of the system to be developed between all partners of the project. In BDD, all test cases are called *acceptance tests* and structured by means of *features* or *feature files*, where each feature can contain several *scenarios*. Each scenario constitutes one test case and is based on the *Given-When-Then* sentence structure. Consider the following scenario:

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

A calculator should be implemented and this particular scenario describes the addition of two numbers. However, in order to execute the scenario, we have to bind the sentences to actual test code. This can be achieved using so-called *step definitions* which are tuples of a keyword (such as *Given*, *When*, or *Then*), a regular expression, and step code. Whenever a sentence (also called *step*) of a scenario matches the regular expression, the step code is executed. The step definitions for the scenario described above are formalized 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
```

For this purpose, we make use of Ruby [7], a flexible general-purpose object-oriented programming language, that puts a particular emphasize on the design of embedded domain specific languages. Furthermore, we use Cucumber [8] as underlying tool that invokes 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==`'.

Consider the first step definition, that matches the first sentence "Given a calculator" from the scenario in (1). Here, a new instance of a class called *Calculator* is created. For the remaining sentences, the sum of two numbers is calculated by invoking the method *add* (*When*-sentence) and the result is checked against the given test value (*Then*-sentence). As can be seen, the step definitions can be re-used for similar sentences since regular expressions are used to match the steps. In this case, the precise numbers in the scenario can be replaced without modifying the step definitions.

After all steps have been matched and the precise parameters have been inserted, the following source code can be obtained for the scenario:

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

The step definitions are written before the implementation phase has been started based on the scenarios given in natural language. Thus, design decisions affecting the structure of the implementation are taken while writing the step code for the step definitions. For the scenario at hand, it has been decided that there is a class called *Calculator* that has two methods *add* and *result*.

During the implementation phase, the test cases are usually executed whenever the implementation changes, preferably using a background task running autonomously. When executing the test cases, steps can only fail due to two reasons, i.e. *syntactic* and *semantic* errors. The first class of errors occur whenever a name cannot be resolved, e.g. when there is no class called *Calculator*, or there are no such methods as *add* or *result* available yet. The second class of errors consists of errors that occur whenever values do not match their expectations. This can only happen in assertions, i.e. step definitions such as the third one given in (1).

After specifying the step definitions, when executing the test cases, they will fail at the first sentence and stop with an error message stating that the name *Calculator* cannot be resolved. This points the developer to create a class *Calculator*. Following this flow for the next two sentences, a code structure that is implied from the code in the step definitions is implemented as follows:

```
class Calculator
  def add(a, b)
  end

  def result
  end
end
```

With this code, the test cases will pass up to the third sentence which is bound to the assertion expecting the return value of the *result* method to equal 9. Consequently, an error message is returned informing about the wrong result since there does not exist a correct implementation for *result* and *add*. As a result, at this point the designer is guided to implement the required functionality:

```
class Calculator
  def add(a, b)
    @result = a + b
  end

  def result
    @result
  end
end
```

This ends the design process for this scenario. As can be seen, the code does not have to be tested as a post-process since the tests were run during the whole implementation process. In fact, the tests have been guiding this process. In the successive steps, more features and scenarios can be added to complete the implementation of the overall system.

Often, one scenario should be checked against several test assignments or corner cases. For example, if the scenario described in (1) should also be applied to other addends, it would be convenient to reuse the same scenario, since the step definitions are already generic. For this purpose, feature files allow the specification of *scenario outlines* which provide parameterized scenarios enriched with *example tables* allowing for several test assignments. A scenario outline for the scenario in (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

In the next section the proposed flow for circuit design and verification based on BDD is introduced.

III. Proposed Flow

Based on the BDD flow that has been described in the previous section, the basic idea for the proposed flow is

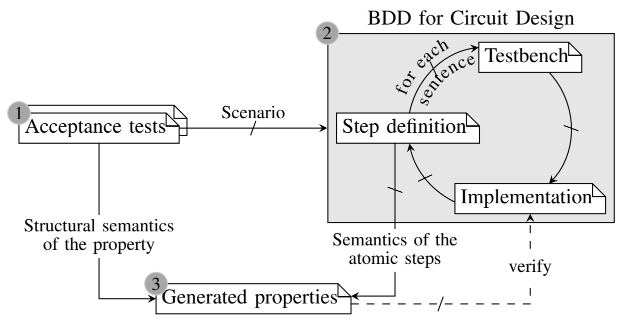


Fig. 1. Proposed Flow

introduced in this section, while details are outlined in the next section. Our proposed flow consists of two main contributions, i.e. (1) customizing the BDD flow such that it is suitable for circuit design by handling the specifics that only exist in HDLs and (2) generalizing the written test code to properties that are suitable for applying formal verification techniques to the implemented design.

A. BDD for Circuit Design

The main steps of our flow are depicted in Fig. 1. Following the BDD process, in Step 1 the features of the hardware (or hardware components) are described by *acceptance tests* in natural language using the *Given-When-Then* sentence structure. This step does not differ from the conventional BDD flow.

These acceptance tests should now be used to create a circuit design using Verilog. The design process can be applied to similar languages such as VHDL accordingly. For the actual creation of the system, we enter the gray area of Fig. 1 (Step 2). Here, the step definitions, the testbench, and the implementation are developed iteratively (emphasized by the circle in the figure). The first specific that arises in the implementation using an HDL is the existence of a separate testbench that allows for executing the circuit design. Recall, that during this process each addition or modification immediately causes the test cases to be executed according to the BDD process with the objective that the acceptance tests pass. If this execution fails, code is missing and additional iterations are necessary. By creating the *step definitions*, essential fragments of sentences are mapped to code. In the beginning these include the definition of I/O (input and output signals) for the considered hardware component(s) while writing the first lines of *testbench* code within the step definitions. In this situation the next task is to add all necessary I/O to the *implementation*. This corresponds to fix all syntactical errors during the execution of the test cases. To further get the test cases also semantically correct, afterwards the respective functionality needs to be implemented. Whenever the implementation is modified, the testbench has to be changed accordingly.

Example 1: The following scenario describes the addition operation of a simple arithmetic logic unit (ALU).

Background:
  Given module alu
  Given testbench alu_test

Scenario Outline: *Adding*
  When I add the numbers <a> and <b>
  Then the output is the sum of <a> and <b>
  Examples: (3)

a	b
2	5
3	4
0	6

This scenario additionally consists of a *background section*, that is used to specify the name of the implementation and the testbench. The step definitions in this background section are predefined and do not have to be specified by the designer.

Given this scenario the following step definitions are written in Ruby where the Verilog test code is embedded in the step code using a *here document*, i.e. a string enclosed by '`<<VERILOG`' and '`VERILOG`':

```
When /^I add the numbers (\d+) and (\d+)$/ do |a,b|
<<VERILOG
  in1 = a;
  in2 = b;
  op = 0;
VERILOG
end

Then /^the output is the sum
       of (\d+) and (\d+)$/ do |a,b|
<<VERILOG
  $assert(out === a + b);
VERILOG
end
```

As can be seen, the designer takes decisions about inputs and outputs of the ALU module. The step code is Verilog code, however it cannot be instantiated directly but requires a testbench body that encloses the test code. This testbench body is written in the next step as follows.

```
module alu_test;
  reg [7:0] in1;
  reg [7:0] in2;
  reg op;
  wire [7:0] out;

  alu a(in1, in2, op, out);

  initial begin
    $yield;
  end
endmodule
```

In the testbench the user takes further decisions about the bit-width of the input and output signals as well as the signature for the actual module, i.e. the order in which the input and output signals are provided. The '`$yield`' command is no Verilog command but a placeholder which is substituted by the Verilog test code that is extracted from the step code in the BDD flow (see step definitions above). Based on this, finally the implementation is created which can then be checked against the acceptance tests. ▮

B. Generalization of Test Code

After all acceptance tests have passed, an implementation is available that fulfills all requirements that have been specified by them. However, due to their nature the acceptance tests can never cover a scenario exhaustively. As a result, only a subset of test patterns are applied via the example table of a scenario outline. We propose to *generalize* the acceptance tests by automatically generating PSL [9], [10] properties. Thus, the scenario can be verified thoroughly using the existing state-of-the art algorithms for formal verification. In order to obtain the PSL properties, first the structural semantics of a scenario is used by mapping the *Given-When-Then* sentence structure to an implication property. On the other hand the expressions of the property (statements in the antecedent and consequent) are formed by using the "glue code" and the step code of the step definitions. There, the respective variables as well as symbolic relations can be extracted due to the organization of the testbench code coming from scenario outlines.

Example 2: From the scenario outline and the respective step definitions in Example 1, the following property can be generalized automatically.

```
property adding =
  always ( op == 0 )
  -> ( out == in1 + in2 );
```

The idea is that test code from step definitions in the *When*-sentences and *Then*-sentences is part of the antecedent and consequent of the property, respectively. To resolve the relation between variables appearing inside the antecedent and the consequent, the *glue code*, i.e. the part of the scenario outline and the step definitions that relates the variables and placeholders such as <a> and <b> to the parameters inside the step code, is used. ∎

In the next section the implementation of our flow is detailed.

IV. Implementation

The last section provided the general idea of how the BDD flow can be customized to support the specifics that are given due to the usage of HDLs as well as how the test cases can be generalized as properties. This section provides more details on the implementation of the proposed approach with emphasizing in particular the latter task.

A. Customizing BDD for HDLs

When customizing the BDD flow for supporting HDLs, two major issues need to be considered, i.e. (1) the separation of the actual implementation and a special testbench and (2) timing information that arises when simulating hardware. For this the following three tasks have to be performed.

1) Setting up Implementation and Testbench: The scenario syntax for BDD allows for a special background section in a feature file that consists of steps which are executed before each scenario in that feature file. The testbench is prepared and consists of a skeleton in which the actual test code from the step definitions is inserted. The primary tasks are instantiating the module and connecting the test signals to the signals of the module. These test signals can then be assigned values in the step code. It is possible to reuse this testbench skeletons in multiple feature files.

2) Timing: When designing hardware with HDLs a cycle-accurate simulation is of high importance. For this purpose, the designer should add precise timing information to the test code in the testbenches by prefixing a statement using '$\#t$' where t denotes the cycle after which the following statements are executed. Due to the re-use of the step definitions the respective step code can be assembled in different ordering inside the testbench. Therefore, no absolute timing can be specified. However, relative timing information is allowed and implemented using an internal counter that stores the current time initialized with 0. The timing information is then added automatically to each Verilog statement when assembling the step code. The internal counter can be incremented inside a scenario using two different ways. First, the predefined step "I wait t cycle(s)" can be used, which will increment the internal counter by t:

```
When /^I wait (\d+) cycles?$/ do |cycles|
  wait(cycles.to_i)
end
```

Second, the function '`wait`' is provided by our API and can also be called inside a user-defined step definition, e.g.:

```
When /^the output is (\d+)
      after one cycle$/ do |result|
  wait(1)
<<<VERILOG
  output = result;
VERILOG
end
```

3) Executing the Testbench: In our proposed flow, the tests are executed as follows. For each scenario in a feature file, first the module and the testbench skeleton are determined from the background section. Assuming that a scenario consists of n sentences (steps), n testbenches are created that span over the first $1, 2, \ldots, n$ sentences of the scenario. This is required to find the location of the error in a failing scenario, i.e. the sentence that causes the test to fail. Before the step code is inserted into the testbench skeleton, the timing information is added to each inserted statement. The considered module and assembled testbench are then compiled using a Verilog compiler and after that executed using a Verilog run-time engine. In our experiments, we are making use of the Icarus Verilog suite [11].

Alternatively, other Verilog frameworks can be integrated, in fact, using an interpreter the steps can actually be executed in an iterative manner without assembling n testbenches for one scenario.

B. Generalizing Properties from Test Code

For each scenario outline in the feature file, a property is generated as follows.

Algorithm P (*Property Generation*). Given a scenario outline, this algorithm generates a property from it.

P1. [Resolve dependencies.] Since inputs and outputs need to be related, the parameters inside the step code must be replaced by the placeholder variable from the scenario outline.

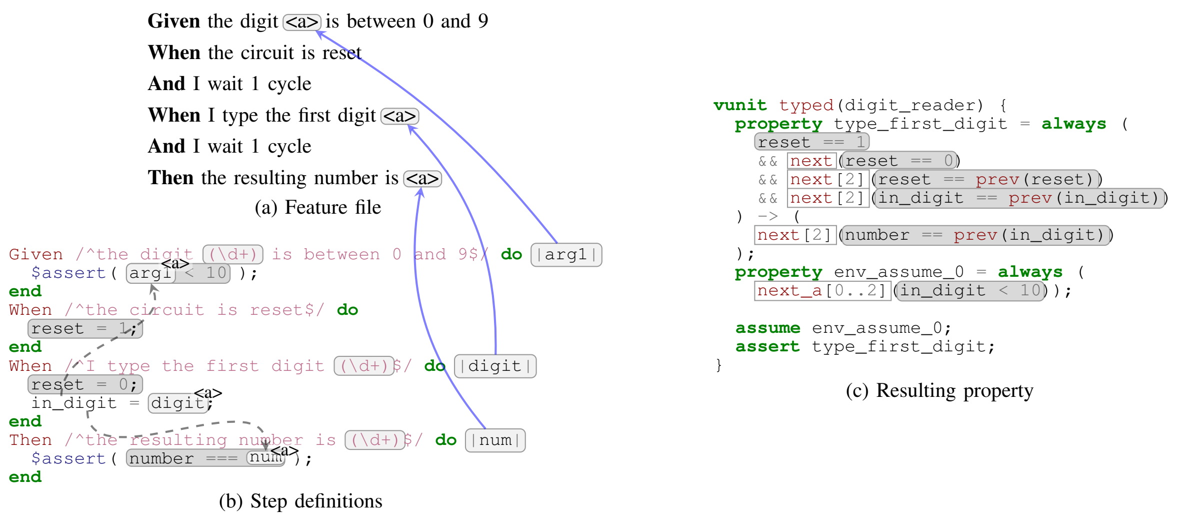
P2. [Term rewriting.] Inputs and output signals are related by the placeholder variable. Thus, expressions containing inputs and placeholder variables need to be rewritten such that they are expressed by the placeholder variable.

P3. [Timing.] Timing information from the test code is used to adjust the properties by surrounding input and output signals with '`next`' and '`prev`' temporal operators accordingly.

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. ∎

While Algorithm P briefly sketches the general outline of the proposed approach, the following subsections will explain the individual steps in more detail.

1) Resolve Dependencies: The step code inside the step definitions assigns values to the inputs and checks outputs for expected values. This is usually done in individual step definitions where the respective values are passed as parameters to the step code. In order to create properties from the test code, it is necessary to relate the inputs to the outputs. However,



Given the digit <a> is between 0 and 9
When the circuit is reset
And I wait 1 cycle
When I type the first digit <a>
And I wait 1 cycle
Then the resulting number is <a>

(a) Feature file

```
Given /^the digit (\d+) is between 0 and 9$/ do |arg1|
  $assert( arg1 < 10 );
end
When /^the circuit is reset$/ do
  reset = 1;
end
When /^I type the first digit (\d+)$/ do |digit|
  reset = 0;
  in_digit = digit;
end
Then /^the resulting number is (\d+)$/ do |num|
  $assert( number === num );
end
```

(b) Step definitions

```
vunit typed(digit_reader) {
  property type_first_digit = always (
    reset == 1
    && next(reset == 0)
    && next[2](reset == prev(reset))
    && next[2](in_digit == prev(in_digit))
  ) -> (
    next[2](number == prev(in_digit))
  );
  property env_assume_0 = always (
    next_a[0..2](in_digit < 10));

  assume env_assume_0;
  assert type_first_digit;
}
```

(c) Resulting property

Fig. 2. Implementation

this is not possible from the step definitions alone since each step is considered independently from the other ones.

In order to resolve the dependencies between the step definitions to relate the inputs to the outputs, the placeholder variables such as <a> and <b> that appear in a scenario outline are used. Together with precise values given by means of rows in the example table, these variables correspond to selected test patterns that are assigned to the scenario. Since the same placeholder variable names are used to target the same inputs and outputs in the scenario, they allow for resolving the dependencies in the step code. For this purpose, first the parameters in the step code are replaced by the placeholder variable name.

Example 3: Consider the scenario outline and step definitions in Fig. 2a and Fig. 2b. As can be seen, the parameters '`arg1`', '`digit`', and '`num`' are all substituted by the variable <a>. This is indicated by the blue arrows that connect the parameter with the respective placeholder in the scenario. Having this dependency, the parameter occurrences inside of the step code can be annotated by the respective placeholder variable. ▮

All information necessary to resolve these dependencies are highlighted using the light gray boxes.

2) Term Rewriting: Once all dependencies have been resolved, all statements in the step code of the form $i = e$ (where i is an input signal and e is an expression that consists of a placeholder variable p) are first rewritten and then used to relate the outputs to the inputs. That is, the expressions are transformed in terms of p such that its right-hand side can be used for substitution in the output expressions of the property. More precisely, first the statement $i = e$ is transposed to the statement $p = e'$ with p being the placeholder variable in e and e' such that $(i = e) \equiv (p = e')$. Then, all placeholder occurrences in output expressions can be replaced by e'.

Example 4: In Fig. 2b the statement '`in_digit = <a>`' (i.e. $i = e$) is such an expression. It is first transposed to '`<a> = in_digit`' (i.e. $p = e'$), and afterwards all occurrences of <a> inside statements that contain output signals can be substituted using this expression. In Fig. 2b, this is illustrated by the dashed arrows. Hence, the step code inside the first step definition is translated to '`$assert(in_digit < 10)`' and for the last step definition to '`$assert(number === in_digit)`' accordingly. ▮

There are some special cases that need to be considered when applying this technique. When there are more statements assigning an expression to an input signal always the last one that would be invoked at test execution is used for substitution, since the last statements override previous input assignments.

3) Timing: For timing consideration each statement in the step code is annotated using a current time t starting from $t = 0$ and incremented accordingly whenever this internal counter is adjusted using the appropriate API calls (see Section IV-A2). When assembling the property this timing information is translated into respective '`next`' and '`prev`' temporal operators. Note that this information also needs to be preserved when inserting code for relating the output signals to the inputs. If the input has been assigned in a previous step, the substituted code is surrounded by a respective '`prev`' statement.

Example 5: In Fig. 2c all timing information has been highlighted using white rectangles. As can be seen, the output statement that is asserted using the *Then* sentence is executed at $t = 2$, however the input '`in_digit`' is assigned one cycle before. All this information has been taken into account in the consequent of the resulting property. ▮

4) Test Semantics: When executing a test case, the signals are assigned and asserted imperatively and also changing the internal timing counter does not change the values of signals unless explicitly specified. However, this imperative semantics is not implicitly considered in properties for verification and as a result these test semantics need to be ensured explicitly. For this purpose, expressions for signals that do not change are added to the antecedent.

Example 6: For the example in Fig. 2b the additional expressions are given in the last two lines of the antecedent. Since '`reset`' and '`in_digit`' do not change anymore after the first cycle, expressions ensuring the test semantics are added accordingly. ▮

System requirements:
1) The typed numbers are shown on a display.
2) The price of a product is displayed after the numbers are typed.
3) The current price of a product can be changed by service personnel.
4) After the price is paid the product can be obtained.
5) Service mode uses the product numbers 1-19.
6) Product numbers range from 20 to 60.
7) Prices can vary from 0 to 200 cents in intervals of 10.
8) The machine accepts coins 10, 20, 50 for payment.
9) The machine return coins 10, 20.

Fig. 3. Requirements for a vending machine

5) Specifying the Domain of Test Patterns: When writing tests using a scenario outline some test patterns are given through the example table. However, when generalizing test cases to properties all possible assignments to the considered input signals that appear in the test case are assumed. As a result, it is necessary to restrict the domain of the input signals albeit the fact that all test patterns in the example table adhere this constraint already. This additional information that restricts the domain of test patterns can be specified using *Given*-sentences and therefore integrates seamlessly into the BDD flow.

Example 7: Consider again the example in Fig. 2. The scenario includes a *Given*-sentence in line 1 that restricts the digit <a> to be entered to the domain $[0, 9]$. In order to execute the test a step definition has to be specified for that sentence which is done in the first three lines in Fig 2b. ▮

The *Given*-sentences are then transcribed to an independent property. Because those sentences describe a global restriction to the inputs, the property is assumed instead of asserted.

Example 8: In Fig. 2c the presented verification unit contains a second property *env_assume_0* which restricts the possible values of the input *in_digit* to numbers in the domain $[0, 9]$. ▮

V. Illustrative Example

In this section we demonstrate the new flow which has been implemented on top of the *Cucumber* tool [8] in Ruby. Our approach was applied to design parts of a vending machine with system requirements as shown in Fig. 3. Due to page limitation we only describe one scenario in detail. In particular, we present the application of the approach for the third requirement *The current price of a product can be changed by service personnel*. Following the BDD approach several scenarios are now developed for this requirement with one of them being

"When the price of the product is changed by a service technician to a new price and the respective product is requested by a customer afterwards, then the shown price should be the new price that was given before." (4)

As a first step this specified acceptance test is rewritten to match the *Given-When-Then* template by identifying the atomic parts of the scenario, e.g. the word '*afterwards*' indicates that the request of the customer happens in a following cycle. We use the predefined step "I wait *t* cycle(s)" to express timing constraints (cf. Section IV-A2). This results in the following scenario:

Scenario: *Change price and request the same product*
When a service technician wants to change a price
And a product <product> and a price <price> is given
And I wait 1 cycle
And the same product is requested by a customer
And I wait 1 cycle (5)
Then the price of <product> is <price>
Examples:

product	price
21	60
34	70

Having this scenario it can be executed using the *Cucumber* tool which first informs about both a missing module and testbench. For this purpose a *background section* is added to the feature specification including the *memory* module as the implementation to consider, since it saves the prices to specific product numbers:

Background:
Given module memory (6)
Given testbench test_memory

The sentences of the scenario are bound to testbench code by means of step definitions and in particular this testbench code implies naming for inputs and outputs of module *memory*:

```
When /^a service technician wants to
       change a price$/ do
<<VERILOG
  set = 1;
  reset = 0;
VERILOG
end

When /^a product (\d+) and a price (\d+)
       is given$/ do |arg1, arg2|
<<VERILOG
  product = arg1;
  price = arg2;
VERILOG
end

When /^the same product is requested
       by a customer$/ do
<<VERILOG
  set = 0;
  reset = 0;
VERILOG
end

Then /^the price of (\d+) is (\d+)$/ do |arg1, arg2|
<<VERILOG
  $assert(price_out === arg2);
VERILOG
end
```

Since the test code cannot be simulated independently, a testbench is required. The testbench body is written as explained in Section III, i.e. an instance of the desired module *memory* is created, the bit-widths of inputs and outputs are set and finally the '`$yield`' placeholder is inserted which will be replaced with the test code from the step definitions during execution:

```
module test_memory;
  reg[5:0] box;
  reg[7:0] price;
  reg set, reset;

  wire[7:0] price_out;

  memory mem(box, price, set, reset, price_out);

  initial begin
    $yield;
```

```
 1 module memory(
 2   input[5:0] product,
 3   input[7:0] price,
 4   input set, reset, clk,
 5   output reg[7:0] price_out
 6   );
 7
 8   parameter ITEMS = 40;
 9   parameter SERVICE = 20;
10
11   reg[7:0] map[0:(ITEMS)];
12
13   always @(posedge clk) begin
14     if (!reset)
15       if ((product >= SERVICE)
16             && (product < 60))
17         if (set) begin
18           map[product-SERVICE] = price;
19           price_out = price;
20          else
21           price_out = map[product-SERVICE];
22       end else
23         price_out = 0;
24     else begin
25       //initialize memory ...
26     end
27   end
28 endmodule
```

Fig. 4. Implementation of memory module

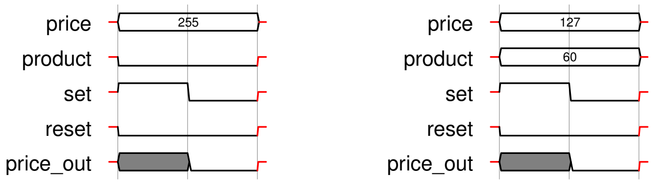
```
 1 vunit service_mode(memory) {
 2
 3   property get_price = always (
 4     set==1
 5     && reset==0
 6     && next (set==0)
 7     && next (reset==0)
 8     && next (product==prev( product ))
 9     && next (price==prev( price ))
10     && next[2](set==prev( set ))
11     && next[2](reset==prev( reset ))
12     && next[2](product==prev( product ))
13     && next[2](price==prev( price ))
14   ) -> (
15     next[2](price_out==prev[2]( price ))
16   );
17
18   //verification-directive:
19   assert get_price;
20 }
```

Fig. 5. Generalized property for the memory module

```
    end
endmodule
```

Given the testbench body, the scenario can be executed using the *Cucumber* tool and fails since no implementation has been created so far. However, the reasons for the failing test will guide the designer in what to implement next such that all steps in the scenario pass the test. The implementation of the module *memory* is shown in Fig. 4. While implementing the module, we insert a new module input for the clock signal, that consequently needs to be inserted into the testbench.

Afterwards, a property is automatically generalized from the scenario by means of the techniques that have been described in Section IV-B. The generated property is depicted in Fig. 5 and generalizes the behavior described in (5), i.e. the behavior is always valid under the given constraints formulated in the test cases. More precisely, if the service mode is enabled in the first cycle (line 4) and is not enabled in the second cycle (line 6) while the price and the product number never changes, then the result should be the price from the first cycle (line 15). We extended the *Cucumber* tool such that the property checker can be called directly via the command line interface as an option.



(a) First counter-example (b) Second counter-example

Fig. 6. Trace view of counter-example

Running the acceptance test (5) with the current implementation all tests pass, however the property fails with the counter-example shown in Fig. 6a. It can be seen that the value 255 is assigned to the input *price*, but considering the system requirements, prices only range from 0 to 200. As a result, it is necessary to restrict the domain of the input signal *price* using the *Given*-sentences as described in Section IV-B5. From the counter-example and the design requirements the subsequent additional sentence to the scenario yields:

Given the price <price> is between 0 and 200 (7)

For this sentence, also a step definition with test code has to be created:

```
Given /^the price (\d+) is between 0
                and 200$/ do |arg1|
<<VERILOG
  $assert( arg1 < 201 );
VERILOG
end
```

Note that the step code of the step definition is executed when running the test cases but unlikely to fail since it targets values that are not present in the example table for the scenario outline. Once the *Given*-sentence has been added to the design, the resulting verification unit will contain another property that describes the restrictions on the input signals.

```
property env_assume_0 = always (
  next_a[0..2] (price < 201)
);

assume env_assume_0;
```

This property ensures that for each considered time step the price is always less than 201.

After successfully implementing the step definition for the *Given*-sentence, now the property does not hold because of another domain restriction problem regarding the product number range. For this restriction another *Given*-sentence is added to the scenario analogously.

When our approach is run with both *Given*-sentences and thus the correct restrictions for all input signals, the property fails due to an implementation error. The counter-example presented in Fig. 6b shows that product is assigned the value 60. Consider the sixth system requirement where the product number 60 is defined as a limit value. The module *memory* presented in Fig. 4 uses this limit in lines 15 and 16 for conditional execution. If the number is not in the given range, the module returns a price of 0, which *price_out* is assigned to. Due to a corner case, the wrong value is assigned.

This is because of the erroneous condition, that does not consider 60 as permitted value (`product < 60`). But since the requirements state that 60 is also a valid product number, the second part of the if-condition should be `product <= 60`. After correcting the if-condition the property holds.

As can be seen, applying behavior driven development for the design of hardware systems leads to a different design flow in which the tests play a central role. The execution of test cases guides the designer in which steps are to be taken next in the implementation. However, although all tests pass eventually, using property checking additional design bugs can be found. Since the properties can be generalized automatically from the scenarios and respective step code, the designer does not have to write the properties manually but can think in terms of test cases, which are more intuitive.

The considered parts of the vending machine design have been specified by means of 14 scenarios, which resemble 51 test cases through their respective example tables. This resulted in 14 properties and 41 domain restrictions needed to be applied. Using the generated properties, we were able to find 6 major implementation errors in the design. Using the new flow the greatest benefit has been the continuous feedback of the property checker, which helped to create a "good" design by considering every possible execution path.

VI. Related Work

To the best of our knowledge the flow and the presented implementation is the first application of BDD for the development of circuits. Moreover, no integration with formal verification – as supported by our test case generalization technique – has been presented. However, TDD for hardware systems has been considered in the past, e.g. based on C++ for embedded systems in [12]. Furthermore, the topic is heavily discussed in many blog posts [13]. SVUnit [14] incorporates agile programming techniques for SystemVerilog to enable TDD by means of unit testing.

The generalization of tests to a formal specification has been proposed in [15]. This work considers Java as target language and the specification is checked by generating runtime assertion checkers in JML. The approach does not facilitate an automatic generalization of tests, instead it is based on user knowledge. Automatic generation of properties has been described in [16]. However, in that work, properties and constraints are generated from *Production Based Specification*, which is a formal specification language based on regular expressions. On the contrary, often the opposite case in which test cases are generated from properties has been intensely considered, see for instance [17], [18].

VII. Conclusions

Based on the success of BDD for software development, we proposed a first implementation on how this agile technique can be applied to chip design using HDLs such as Verilog. For this purpose, we carefully handled the specifics that arise in the design using HDLs and suggested ideas of how they can be integrated in modern BDD tools. Furthermore, a technique has been presented that allows for the generalization of tests as functional properties suitable for formal verification. The new flow enables a test driven development process that guides the designer in the implementation phase while continuously giving feedback in every step of the design flow. Based on the generated properties the design is automatically verified against all possible test patterns and not only to manually specified ones. This allows for finding bugs that are only observable for certain input assignments.

We implemented the proposed approach on top of the *Cucumber* tool and demonstrated it by means of an illustrative example. In future work we want to consider how more advanced timing that is usually found in complex properties can be lifted up to the level of acceptance tests and investigate its application to designs at larger scale. In addition, we plan to automatically analyze the counter-examples to understand the cause of error, i.e. whether the failing of the property is caused by too weak restrictions of the inputs or an actual implementation error. From a higher perspective, we want to integrate natural language processing techniques to automate BDD as proposed in [19]. Moreover, for correctness and efficiency the concept of completeness driven development [20] is essential for our flow.

Acknowledgments.

This work was supported by the German Research Foundation (DFG) within the Reinhart Koselleck project DR 287/23-1.

References

[1] K. Beck, *Test Driven Development. By Example*. Amsterdam: Addison-Wesley Longman, Nov. 2003.

[2] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas, "Manifesto for agile software development," 2001. [Online]. Available: http://www.agilemanifesto.org/

[3] N. Nagappan, E. M. Maximilien, T. Bhat, and L. Williams, "Realizing quality improvement through test driven development: results and experiences of four industrial teams," *Empirical Softw. Engg.*, vol. 13, no. 3, pp. 289–302, Jun. 2008.

[4] E. M. Maximilien, "Assessing test-driven development at IBM," in *Int'l Conf. on Software Engineering*, 2003, pp. 564–569.

[5] D. North, "Behavior Modification: The evolution of behavior-driven development," *Better Software*, vol. 8, no. 3, Mar. 2006.

[6] R. C. Martin, "The truth about BDD," 2008. [Online]. Available: http://blog.objectmentor.com/articles/2008/11/27/the-truth-about-bdd

[7] D. Flanagan and Y. Matsumoto, *The Ruby Programming Language*. O'Reilly Media, Jan. 2008.

[8] M. Wynne and A. Hellesøy, *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. The Pragmatic Bookshelf, Jan. 2012.

[9] *Accellera Property Specification Language Reference Manual, version 1.1*, http://www.pslsugar.org, 2005.

[10] C. Eisner and D. Fisman, *A Practical Introduction to PSL (Series on Integrated Circuits and Systems)*. Secaucus, NJ, USA: Springer, 2006.

[11] S. Williams. (2012) Icarus Verilog. [Online]. Available: http://iverilog.icarus.com/

[12] M. Smith, A. Kwan, A. Martin, and J. Miller, "E-TDD – Embedded Test Driven Development: A Tool for Hardware-Software Co-design Projects," in *Extreme Programming and Agile Processes in Software Engineering*, Jun. 2005, pp. 1229–1231.

[13] N. Johnson and B. Morrs. (2012) AgileSoC. [Online]. Available: http://www.agilesoc.com/

[14] B. Morris and R. Saxe, "svunit: Bringing Test Driven Design Into Functional Verification," in *SNUG*, 2009.

[15] H. Baumeister, "Combining formal specifications with test driven development," in *XP/Agile Universe*, 2004, pp. 1–12.

[16] M. Jahanpour and O. Mohamed, "Automatic generation of model checking properties and constraints from production based specification," in *Midwest Symposium on Circuits and Systems*, Jul. 2004, pp. 435–8.

[17] M. P. E. Heimdahl, S. Rayadurgam, W. Visser, G. Devaraj, and J. Gao, "Auto-generating Test Sequences Using Model Checkers: A Case Study," in *Formal Approaches to Software Testing*, Oct. 2003, pp. 42–59.

[18] Y. Zheng, J. Zhou, and P. Krause, "A Model Checking based Test Case Generation Framework for Web Services," in *Int'l Conf. on Information Technology*, Apr. 2007, pp. 715–722.

[19] M. Soeken, R. Wille, and R. Drechsler, "Assisted behavior driven development using natural language processing," in *TOOLS (50)*, 2012, pp. 269–287.

[20] R. Drechsler, M. Diepenbeck, D. Große, U. Kühne, H. M. Le, J. Seiter, M. Soeken, and R. Wille, "Completeness-driven development," in *International Conference on Graph Transformation*, 2012.