Insights from agile software development techniques used for hardware development

Darío Macchi
macchi@uni.ort.edu.uy

Abstract. Agile software development techniques are no new approaches in hardware development. However they can be used with formal specification languages for hardware design and verification through the automatic generalization of Behavior Driven Development (BDD) test cases. Although the main reason of using BDD in software development is the increase of the communication between every stakeholder, we think that formal specification languages can add formality to BDD due to automatic property validation techniques. It can be used in some risky domains or just for those features with higher priority in project backlogs.

Keywords: agile software development; behavior driven development; property specification language; formal methods; verification.

1 Introduction

TDD is a design flow paradigm in which test cases are provided as starting point and central elements along the whole design process [1]. Test cases are specified first (challenging traditional way [2][3][4]), and, since no implementation is available, they will initially fail. Based on error messages from failing test cases, the implementation grows incrementally until all test cases pass eventually [5]. Over this design flow paradigm, BDD has been 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 [6][7]. The natural language ensures a common understanding of the system to be developed between all partners of the project [8]. 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 [1]:

```
Scenario: Adding two numbers
  Given a calculator
  When I add the numbers <a> and <b>
  Then I see the result the sum of <a> and <b>
  Examples:
    | a | b |
    | 2 | 5 |
    | 3 | 4 |
    | 0 | 6 |
```

Fig. 1 - Example of a BDD test case written in natural language

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.

The article entitled "Behavior Driven Development for circuit design and verification" propose the use of BDD technique in hardware design and validation. Basically they propose an algorithm to generalize BDD acceptance tests and later convert them into formal properties automatically using PSL (Property Specification Language). This algorithm (although it need more refinement [1]) is a novelty in hardware development, even though Test Driven Development was proposed by other authors to co-design hardware-software projects [9]. From the software development perspective, this proposal initially doesn't seems to practically add benefits for everyday BDD use; the generalization offered by PSL is right now achieved using regular expressions for every sentence (also called step) that lead to the execution of specific step code. Also, software developers use libraries for fake data generation (e.g. Faker [10]) together with factory libraries to build fixtures, scenarios and stubs (e.g. Factory_Girl [11]). However, the formalization of each step can be used in software development to perform automatic validation of each step using PSL tools (something that seems to be difficult to achieve with regular expressions and the aforementioned tools).

2 Viewpoints

The use of agile software development techniques in hardware design is not new. Some authors claim that Test Driven Development (TDD) can be used to co-design hardware-software projects [9]. They chose to adapt the small CppUnitLite [12] developed by Michael Feathers mostly because it minimize memory usage to ensure that it would fit within the embedded system environment. The main issue they tried to solve was that

all tests had to be available for running at all times without using an external development environment [9].

The viewpoint of the article "Behavior Driven Development for circuit design and verification" increase the abstraction level and introduce the concept of formal verification related with an agile software development technique. It states that circuit design and verification can be done using BDD and later generalize test cases to formal properties. The authors split their contribution in two parts.

- First, they customized the BDD flow for circuits, modelling in Hardware Description Language (HDL). As a result each circuit component is created based on the underlying BDD methodology, connecting test cases to natural language specifications.
- Second they present a technique to automatically generalize the BDD acceptance tests, generating for this properties using the Property Specification Language (PSL). These properties are then formally verified iteratively against the developed implementation.

From the software development viewpoint, the application of formal methods in validation and verification (V&V) activities is not new; in the 60s the Hoare calculus, pre/postconditions, invariants and assertions were introduced for proving program correctness [13][14]. Those ideas gradually were introduced in the industrial software engineering as formal methods, e.g. the Vienna Development Method (VDM) developed at IBM [15], Z [16], the JavaModeling Language (JML) [17] and, more recently, the Object Constraint Language (OCL) [18] – which is used to specify constraints on objects in UML diagrams.

Although formal methods in software V&V is not new, using them without losing agility is still a challenge [19]. For this reason, the novelty of the BDD plus formal methods approach and the possible applicability of it in software development is the second contribution made by authors: the generalization of test cases through PSL properties [1]. Due to how test cases are built in natural language (see Fig. 1) in BDD, only a subset of values can be covered by them via an example table. In order to show this, we continue with the example in Fig. 1.

Given the stated scenario written using the Gherkin's Grammar using the Cucumber tool, the next step is to implement each step definition (in Ruby) embedding Verilog test code using a *here document* (a section of a source code file that is treated as if it were directly written to stdin (standard input) [20]. In the example is the string between <<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 the step code is Verilog code it cannot be instantiated directly, so it requires a testbench body to enclose the generated test code.

```
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
```

When the Cucumber code runs inside this testbench it replace the placeholder `$yield` with the Verilog code *written* by the *here document* inside each BDD step definition:

```
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
       in1 = a;
       in2 = b;
       op = 0;

       $assert(out === a + b);
   end
endmodule
```

Then, the generalization is achieved by the automatic generation of PSL properties to later verify them with existing state-of-the-art algorithms for formal verification. In order to obtain the properties they propose to convert the Given-When-Then structure to an implication property. Also 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.

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

The Fig. 2 shows another example of this generalization process from a BDD scenario to a PSL property implementation.

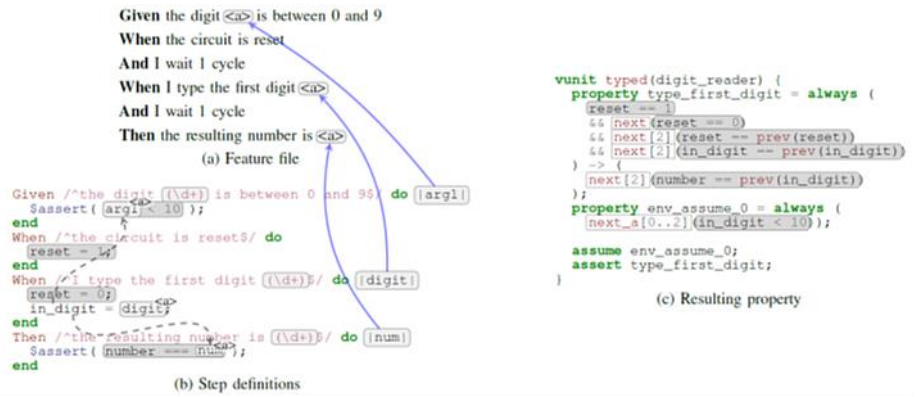

Fig. 2. Another example of generalization process (from BDD scenario to PSL)

In the agile world, BDD is used mostly as a higher-level-of-abstraction technique to improve communication between the stakeholders of a project. So, from that point of view, the use of formalizations is against the spirit of the technique. However, as we stated before, the use of PSL properties is done later, after the "glue code" is generated, so I think it doesn't affect the effectiveness of the technique.

3 Conclusion

Some agile software development techniques seems to be applicable for hardware design and validation. In some cases TDD can help to keep things small from a resource consumption point of view; in other cases, the need of more abstraction power lead to use a higher abstraction technique like BDD. In the last case and because the possible use of hardware that is not upgradeable in sensitive contexts like health (e.g. pacemakers), developers can't trust in finite scenarios. Conversely, they need a way to test every possible value in a formal way and do it automatically.

Software developers, on the other hand, doesn't need this level of trustfulness for every software they develop; always adding this complexity layer may slow down the developer's teams velocity (even more in the case of agile teams). However, in cases where business domain is risky enough, formal methods to ensure that features fit the specifications can be used. I think that in agile contexts, those features that are in the front of backlog (the ones with highest priority) are prone to be verified formally and then those stated as riskier. The proposed algorithm [1] can be a good starting point to do this, although for me is far away to be automatic and more work should be done in this direction.

4 References

[1] M. Diepenbeck, M. Soeken, D. Grose, and R. Drechsler, "Behavior Driven Development for circuit design and verification," *2012 IEEE Int. High Lev. Des. Valid. Test Work.*, pp. 9–16, 2012.

[2] K. Beck, *Test Driven Development: By Example*. 2002.

[3] H. Erdogmus, M. Morisio, and M. Torchiano, "On the effectiveness of the test-first approach to programming," *IEEE Trans. Softw. Eng.*, vol. 31, no. 3, pp. 226–237, 2005.

[4] D. Janzen and H. Saiedian, "Test-driven development concepts, taxonomy, and future direction," *Computer (Long. Beach. Calif).*, vol. 38, no. 9, pp. 43–50, 2005.

[5] D. Sundmark and S. Punnekkat, "Impact of Test Design Technique Knowledge on Test Driven Development: A Controlled Experiment," pp. 138–152, 2012.

[6] D. North, "Introducing BDD," *Better Software*, 2006.

[7] D. Chelimsky, *The RSpec Book: Behaviour Driven Development with RSpec, Cucumber, and Friends (The Facets of Ruby Series)*. Pragmatic Bookshelf, 2010.

[8] John Ferguson Smart, *BDD in Action*. 2014.

[9] M. Smith, A. Kwan, A. Martin, and J. Miller, "E-TDD - Embedded test driven development a tool for hardware-software co-design projects," *Lect. Notes Comput. Sci.*, vol. 3556, pp. 145–153, 2005.

[10] B. Curtis, "Faker." 2008.

[11] Thoughtbot inc., "Factory_Girl." 2008.

[12] M. Feathers, "Cpp Unit Lite." 2004.

[13] C. A. R. Hoare, "An axiomatic basis for computer programming," in *Communications of the ACM*, 1968.

[14] R. Floyd, "Readings in Artificial Intelligence and Software Engineering," C. Rich and R. C. Waters, Eds. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1986, pp. 331–334.

[15] C. B. Jones, *Systematic Software Development Using VDM (2Nd Ed.)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1990.

[16] J. M. Spivey, *The Z Notation: A Reference Manual*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1989.

[17] G. T. Leavens, A. L. Baker, and C. Ruby, "Behavioral Specifications of Businesses and Systems," H. Kilov, B. Rumpe, and I. Simmonds, Eds. Boston, MA: Springer US, 1999, pp. 175–188.

[18] J. Warmer and A. Kleppe, *The Object Constraint Language: Precise Modeling with UML*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[19] H. Baumeister, "Combining Formal Specifications with Test Driven Development," *Extrem. Program. Agil. Methods - XP/Agile Universe 2004*, vol. 3134, pp. 1–12, 2004.

[20] "Here Document." [Online]. Available: https://en.wikipedia.org/wiki/Here_document.