2017 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE

Specification by Example for Hardware Design and Verification

Jussi Mäkelä, Verranto AS, Trondheim, Norway (jm@verranto.com)

Abstract—A specification is a key piece of work that contributes towards the successful delivery of the design and verification effort within project timescales. Specifications are most commonly written using natural language, the style and format of which can result in incomplete definitions and/or suffer from poor accuracy. Natural language specification can be costly to maintain and seen as a low priority if timescales are tight, which can result in outdated or incorrect material by the end of the project. Even well written and well maintained specifications can be misinterpreted causing delays and bugs. This paper presents how principles of Specification by Example can solve these common specification issues and how acceptance test-driven development can be utilised within a test automation framework to document hardware accurately using tests and use cases as examples together with model code.

Keywords—Specification, Test, Test Driven Development, Acceptance Test Driven Development, Specification by Examples

I. INTRODUCTION

In a traditional waterfall development process, requirement and design specifications are key pieces for a successful delivery. By definition, the specification should describe the design precisely, but often fails to do so. Incomplete and changing specifications have been reported to be one of the largest factors that negatively impact project execution [1]

A significant part of project execution time is spent shaping the perfect specification. [2] The time spent developing the specification is justified by its use as the golden reference during development as well as being used as part of a product reference post tape out.

Software projects have adopted agile development processes, such as ‘test-driven development’ and ‘specification by example’, that aim to reduce the time spent on the specification while still producing effective and up to date documentation. Time reduction is achieved by encouraging the all functions of the team to work collaboratively to define test cases or examples that capture the functionality. The resultant output is a living specification that is progressively refined as the project is executed.

In a ‘Test Driven Development’ approach, developers and test engineers create the test cases together. Using an iterative flow, test cases are defined after which design development aims to satisfy the test cases. The ‘Test Driven Development’ approach has been shown to achieve better productivity and more consistent quality results in software projects. [3]

Test Driven Development focuses on how functionality is implemented but still heavily relies on the requirement specification when capturing test cases. In contrast, Specification by Example, which is also called ‘Behaviour Driven Development’ or ‘Acceptance Test Driven Development’, is also a test first approach but it focuses more on capturing behaviour. Test cases act as examples and are used to provide intent and detail, so that the same information can be used both as a specification and a business-oriented functional test. Team members across all functions participate in creating the golden reference. Any additional information discovered during development, such as clarification of functional gaps, missing or incomplete requirements, or additional tests, are added to golden reference. The successful application of Specification by Example has been shown to significantly reduce development iterations in software projects that leads to less rework and higher product quality. [4]

The deployment of test driven development in hardware projects has yet to see detailed documented results. However anecdotal evidence shows similar results in HW projects for both design and verification development. When the HW development team uses a TDD approach with appropriate test cases leading development it results in higher quality in a shorter time compared to traditional development flow where verification happens independently and after RTL development.

Since practices of Test Driven Development have shown encouraging results when used for hardware development, this paper focuses on how Specification by Example could be adopted for hardware development. The paper demonstrates how examples can be used to improve hardware specifications. It also demonstrates how a software framework can be integrated into a simulation environment to enable executable specifications. By following these approaches, this paper suggests that results similar to those seen in software projects should be achievable also in hardware development projects.

II. Common Issues in Specifications

For a point of reference for the specification issues discussed in this paper, we use a theoretical division unit. The division unit has been chosen as the design unit for ease of reference, but components of any complexity could be substituted for use with the approach that is discussed.

Table I. Example specification of Divide module

The design implements a long division for 2 unsigned integers. It divides a number (N) with a second number (D) to give the result (Q), the remainder (R), and a rounding bit (U). The rounding bit (U) is set to 1 if the fraction of the division is greater or equal to 0.5, otherwise it is set to 0.

The long division shifts gradually from the left to the right-hand side of the dividend, subtracting the largest possible multiple of the divisor at each stage. The multiples become the digits of the quotient and the final difference is the remainder.

If the remainder is larger or equal to half of the divider the rounding bit will be set.

```
Q = 0
R = 0
for i = WIDTH-1 ; i >= 0 ; i-- {
  R = R << 1
  R[0] = N[i]
  if R >= D {
    R = R - D
    Q[i] = 1
  }
}
U = (R << 1) >= D ? 1 : 0
```

A. Specifies Implementation but not Intention

Software development teams often receive work packages as a list of requirements rather than a definition of the problem the requirements aim to solve. In extreme cases, this leads teams to blindly accept suggested solutions and then struggle to implement them. Successful teams seek a greater understanding of the problem to identify ultimate goals and derive the design from those. [4]

Loss of intent in the design specifications is also a common issue in hardware development projects. When such specifications are used as a reference for a verification plan, there is a high risk that the chosen strategy focusses too much on the requirements of the implementation but misses nuances lost with the intention. As a result, the verification effort can fall short, which places more pressure on validation to check that the implementation meets the intention.

In the example specification in the table I there is an explicit statement that long division is used, yet it does not capture any reasoning for this. If such statements are not challenged they could result in delays late in the project cycle for both design and verification. For design, delays could be in the form of major rework for functional refinement or PPA optimisations. For verification, delays can be caused by test environments that are

too tailored towards a given implementation such that they also need significant work as the design evolves during development.

For this example, the requirement for the block could be defined as:

> *"Calculate result Q = N / D and remainder R = N % D for positive integers."*

In the example, the implementation defines an exact match requirement. In more complex cases it may be appropriate to define some form of accuracy or precision to allow for better PPA results; for example:

> *"Calculate N / D for positive whole numbers so that result is within +/- 0.5 of the mathematically correct result."*

Capturing the precision requirement would allow verification to check against an accurate model and tolerate different design choices. Also, the model could benefit from implementing an accurate algorithm with a parameterised error margin for exploration purposes.

B. Missing Important Details

In cases where the intention behind the requirement is not captured, it is not uncommon that important details such as corner cases are left out as well. In the case of the division unit example, the description does not describe if D is 0 or both N and D are 0.

C. Use of Pseudocode

The use of pseudocode to describe functionality is a very common practice in textbooks, scientific publications, and design specifications. Natural language is ambiguous and context dependent, and as such does not fit well for describing algorithms or complex hardware systems, whereas code is a very natural choice for that. Pseudocode does not obey any syntax rules and thus, depending on the writer, pseudocode may vary in style.

In the division unit example, the pseudocode is presented in a near-exact imitation of a real programming language. Closely matching the style of the implementation language can be a very hazardous practice. Pseudocode can end up as a golden reference that may not be validated, and copied and transformed directly to executable form into the model and RTL. If there are any errors or inaccuracies in pseudocode they may end up in both model and RTL causing undetectable bugs.

III. SPECIFICATION BY EXAMPLE

"Specification by example (SBE) is a collaborative approach to defining requirements and business-oriented functional tests for software products based on capturing and illustrating requirements using realistic examples instead of abstract statements." [5]

The principles of Specification by Example, which comprise test cases as examples, do not really solve issues A and B presented in section II. Even if examples are used, they might be too specific and fail to capture the intent, or miss important corner cases. No matter the approach, there is always a risk of an incomplete or inaccurate specification. However, the collaborative approach of SBE engages multiple team members from different function early in the process. Successful teams are encouraged to challenge why things are done in a specific way and propose alternative solutions. [4]

A software development case study reported good results from small workshops between developers, testers and business analysts during which features were captured using test cases. The benefit of combining groups in this way comes from the mixture of different backgrounds and experience that results in a more complete definition. [4]

This author suggests that a workshop held between designer, verification, and modelling engineers would produce similar results for a hardware team. If we return to the example in section II, a model engineer could easily highlight the need for an actual algorithm, whereas verification engineer could highlight potential corner case issues. The team would capture all information derived from their different perspectives into a specification in form of examples.

If the examples, which are used to illustrate specification, are captured in executable form, they can be used as test cases to validate real code. Validating code with test cases ensures that the code is accurate, functional and is kept up-to-date. The validated reference code can then be used as a code reference in the specification without same risks as in using pseudocode as described in section II. It is essential to capture the test specification at a high level to separate the test data from the implementation. The separation allows the same tests to be run on different implementations with minimal effort. This kind of high level format for test specification is supported by many test frameworks such as the Robot Framework, which is looked at in more detail in the next section.

IV. ROBOT FRAMEWORK

There are many software tools and test automation frameworks for acceptance testing and acceptance test-driven development, such as Cucumber, Robot Framework and Fitnesse. This paper presents the use of the Robot Framework to demonstrate how such a tool can be utilised within hardware development.

The Robot Framework architecture separates the test data from the test libraries and tools. The structural separation of the framework means that the same test data or test description can be run on different implementations. In terms of hardware development, this allows the validation of the model and RTL using the same set of test data.

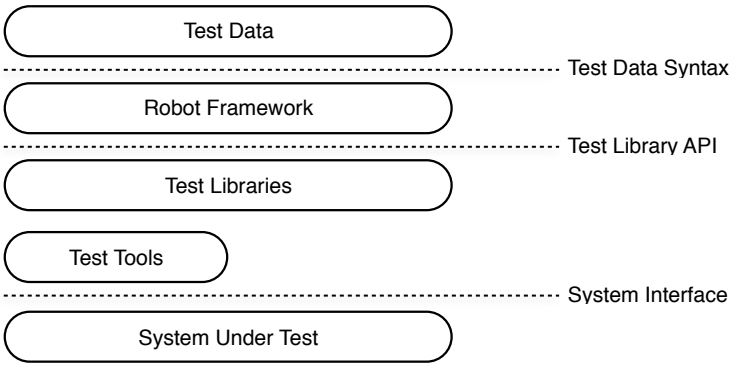

Figure I. High level architecture of the Robot Framework [6]

A. Test Format

The test cases use a keyword testing or data driven methodology. The keyword and test cases combine to form the specification in a logical language that is easy to write in a way that is human readable and understandable while also enabling execution by the Robot Framework to check that the actual implementation gives expected result.

Returning to the division unit example from the previous section, a keyword definition can be captured for ‘Divide’ as shown in the code snippet presented in table II.

Table II. Keyword definition in Robot Framework plain text format

```
*** Keyword ***
Divide
   [Documentation]   Divide operation divides number N with number D which both
   ...               are whole numbers and gives result (Q), remainder (R) and
   ...               round up information (U). U is 1 if fraction is 0.5 or
   ...               larger and otherwise it is 0.
   [Arguments]       ${N}  ${D}  ${Result}  ${Remainder}  ${RoundUp}
   Set input N to ${N} and input D to ${D}
   Perform operation
   The output result should be ${Result}
   And output remainder should be ${Remainder}
   And output round up should be ${Roundup}
```

The keyword definition captures the inputs and outputs for the divide unit example in a human readable and intuitive form. Each set of statements in the keyword definition are mapped by the Robot Framework to stimulus generation and checks. For example the statement “Set input N ...” defines inputs for an API call Divide(N, D), and the statement “Perform operation” makes the call and reads the results. The statement "The output result... And ...” performs the check against the returned results. The API Divide call performs the division operation via the system under test as defined by the framework configuration. For example, the divide operation could be implemented in Python, C-model code, or be returned from a Verilog simulator.

Specification detail is added using test cases that reference the keyword definition. In its simplest form, the test cases can be captured as a test data table which add detail to the specification by defining examples that capture the correct results in different scenarios. The Robot Framework recognises test data tables and executes them as defined with keywords.

The Robot Framework supports multiple formats test data tables like plain text, html and reStructuredText. The code snippet in table III shows two test cases embedded into a reStructuredText formatted document with descriptions in natural language. The reStructuredText format was chosen for this example as it allows formatting, inclusion of code, and it can be rendered to different document formats such as html and pdf. Having output rendering options bridges to project flows that expect more traditional document formats.

Table III. Sample test definition in reStructuredText format

```
================================================================================
Divide
================================================================================

The divide operation divides number N with number D which both are whole numbers
and gives result (Q), remainder (R) and round up information (U). U is 1 if
fraction is 0.5 or larger and otherwise it is 0.

.. literalinclude:: DivisionLibrary.py
 :pyobject: DivisionLibrary.ExactAlgorithm

--------------------------------------------------------------------------------
Examples
--------------------------------------------------------------------------------

When number N is fully divisible by number D then the result should be N/D and
there should be no reminder nor roundup set.

========================  =======  =======  ========  ==========  ============
  Test Case                N        D        Result    Reminder    Roundup
========================  =======  =======  ========  ==========  ============
 No Fraction               255      255      1         0           0
========================  =======  =======  ========  ==========  ============

When number N is divisible by number D so that the fraction is less than 0.5 it
should result a remainder but no roundup.

 ========================  =======  =======  ========  ==========  ============
   Test Case                N        D        Result    Reminder    Roundup
 ========================  =======  =======  ========  ==========  ============
  Fraction less than .5     255      12       21        3           0
 ========================  =======  =======  ========  ==========  ============
```

The separation of the test data from the keyword definition means that the same data set can be executed for different definitions. The keyword captures the implementation detail of specification, or in other words, how the data is used. The data set holds the functional detail of the specification, or in the other words, what the implementation should achieve. The data set should be more stable for same functionality and the separation allows for reuse of this data for different implementations or verification boundaries.

The divide example used in this paper so far shows how the approach can be applied effectively to unit level components. If the test boundary of the example was pushed to that of a top-level CPU, the same test cases could

be used if an alternate keyword definition was provided. For example, “Set input N ...” could be replaced with "Write value ${N} into register R1...". With the alternate keyword definition and the appropriate the mapping API calls, the test data, where the detail is held, can be reused without any alteration.Integrating Robot Framework to a Simulation Environment

B. Integrating Robot Framework to a Simulation Environment

Using the Robot Framework to validate a pure algorithmic C/C++ model is the same as testing any other software codebase. The Robot Framework is implemented in the Python programming language and python integrates well with C and C++. A practical solution for a C/C++ model is to compile the C/C++ code into a library and then define and implement a test API library to map the calls between Python and C/C++.

Hardware designs are realised most commonly using Verilog, VHDL, or SystemC and verified using simulation tools. As the simulation tools cannot execute Python code, the Robot Framework needs to run as a separate, concurrent process from the simulator. The Robot Framework supports testing remote libraries and executables but to be able to do that it needs a test library which handles the test API calls and connect them to external process via a system interface.

One solution to implement the system interface needed for connecting the simulator with the Robot Framework is to use remote procedure calls (RPC). Using RPCs means the test API library can be the same as when testing pure a C/C++ design, but the test library does not include the actual implementation of the function to be tested. Instead, the test library implements a message based communication API using sockets as the channel between the framework and the simulator running in the external process. Communication using sockets and messages is language agnostic and allows the same API to be used with different target environments. The simulation environment wrapping the design to be tested needs to implement a socket server to listen and serve messages.

Messages passed between processes must be serialised into a format that is understood by the sending and receiving sides. By default, the Robot Framework supports remote library calls utilising the XMLRPC protocol, but there is nothing that prevents the use of other protocols.

To demonstrate the integration of the Robot Framework with a Verilog HDL simulator, a C-library was developed to implement socket server to be run on the HDL simulator and provide an interface to Verilog using the PLI/VPI. On the Robot Framework side, the test library and test API were implemented in Python. Framework ‘keywords’ utilised API calls to pass data to and from the simulator. Messages were serialised with the msgpack library. The msgpack library can be found for Python, C, and other languages.

Figure II illustrates the integration between the Robot Framework and a simulation tool. In this setup, when Robot Framework executes test cases, it starts the simulator as an external process and establishes a communication socket to simulator. As the simulator starts, a socket server is created that waits for the messages the Robot Framework will send. Once a message is received it is decoded and triggers a call to the appropriate function or task that performs the required operation; such as, drive inputs or read outputs. Once a result is received from unit under test, the response is sent back to Robot Framework via the unix socket.

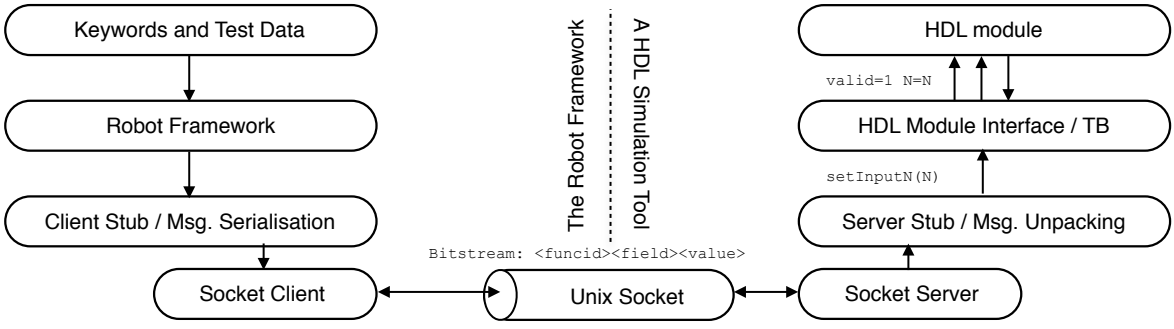

Figure II. The Robot Framework integration to HDL simulator with RPC over unix domain socket

The trial environment described in this paper successfully demonstrated that it is possible to utilise the Robot Framework with a HDL simulator. However, the solution described has issues that need attention if it were to be used in product development. These issues are explained in following section.

C. Issues

Lessons learned from software development show that executable specifications are not well suited to end-to-end testing of complex systems, and are advised against. [4] The definition of end-to-end testing in this instance is the verification and validation that all sub-components of the DUT operate together as intended to deliver against the product requirements, and that it is implemented correctly to allow it to function within the target system as specified by the domain logic of the product.

Hardware designs often include multiple components that talk to each other over different interfaces, as such, any test to exercise the complete design needs to be in the form of an end-to-end test. Keeping this in mind, careful thought is needed as to where executable specifications fit best into the hardware development flow. A viable option is to use test cases or examples to define unit-level areas of functionality and validate their implementations. Standard verification methods then can be utilised to verify the combination and integration of the units. For example, if a design feature can be implemented completely in a task or function, an executable specification can be used to test that feature in isolation leaving higher-level testing to a separate verification environment. By focusing on unit-level verification of features, Specification by Example is not a replacement for traditional verification methods but a complementary solution to improve initial quality.

Even when utilised on individual pieces of functionality, the example implementation described in this paper has flaws. The value gained from the approach comes from executing tests early in the development cycle, but if the approach consumes more time than standard development flows with bug fixing iterations the benefit is lost. The chosen custom solution was simple to implement but included manual work. A time saving optimisation for a more permanent solution would be to autogenerate the RPC library creation. The definition of the client side function call which is passed to the server side should come from single source and the whole connection implemented automatically. For automatic generation, a gRPC framework or an XMLRPC could be better choice than the custom solution used in the trial implementation described in this paper.

V. CONCLUSION

Software testing can be seen as a very similar task to hardware verification: ensure that a product, service, or system complies with a regulation, requirement, specification, or imposed condition. Despite their similar goals, software testing and hardware verification have evolved in quite different directions.

The software development community has widely adopted Agile processes to ensure the right product is built with high quality in a release schedule appropriate to the product. Test Driven Development and Specification by Example are two of the Agile processes that have been shown to improve quality of the final product while having a positive impact on delivery schedules.

This paper has demonstrated, with examples, that ‘specification by example’ with an ‘acceptance test driven development framework’ can be used during hardware development in similar ways seen in software projects to tackle the issues experienced in both flows. With these successful demonstrations, this paper suggests the appropriate deployment of ‘specification by example’ could increase initial quality, which in turn would result in an overall reduction of the product design and verification effort.

ACKNOWLEDGMENT

The author would like to acknowledge key contributions from Robin Hotchkiss of Xtreme-EDA. Without the support and insight from R. Hotchkiss this paper would not have reached its current state.

REFERENCES

1. Standish Group, “Chaos,” Standish Group Report, 1995.

2. https://en.wikipedia.org/wiki/Waterfall_model, read August 6th 2017

3. H. Erdogmus, M. Morisio, and M. Torchiano. On the Effectiveness of the Test-First Approach to Programming. IEEE Transactions on Software Engineering, 31, 2005.

4. Adzic, Gojko (2011). Specification by example: How successful teams deliver the right software. Manning. ISBN 9781617290084.

5. https://en.wikipedia.org/wiki/Specification_by_example, read August 6th 2017

6. http://robotframework.org/#introduction, read August 6th 2017