Received January 31, 2021, accepted February 11, 2021, date of publication February 16, 2021, date of current version March 1, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3059941

Towards Test-Driven Development for FPGA-Based Modules Across Abstraction Levels

JULIÁN CABA[1], FERNANDO RINCÓN[1], (Member, IEEE), JESÚS BARBA[1], JOSÉ A. DE LA TORRE[1], JULIO DONDO[2], AND JUAN C. LÓPEZ[1], (Member, IEEE)

[1]Department of Technology and Information Systems, School of Computer Science, University of Castilla-La Mancha, 13071 Ciudad Real, Spain
[2]School of Electrical Engineering, National University of San Luis, San Luis 4520300, Argentina

Corresponding author: Julián Caba (julian.caba@uclm.es)

This work was supported in part by the Ministry of Economy and Competitiveness (MINECO) of the Spanish Government through PLATINO project under Grant TEC2017-86722-C4-R4, in part by the Regional Government of Castilla-La Mancha through SymbIoT project under Grant SBPLY/17/180501/000334, and in part by the EU's Horizon 2020 Programme through SHAPES project under Grant 857159.

ABSTRACT High-Level Synthesis (HLS) tools help engineers to deal with the complexity of building heterogeneous embedded systems that make it use of reconfigurable technology. Also, HLS opens up a way for introducing, into the development flow of custom hardware components, techniques well known in the software industry such as Test-Driven Development (TDD). However, the support provided by HLS tools for verification activities is limited, and it is usually focused on the initial steps of the design process. In this paper, a hardware testing framework is introduced as an enabler for effortless on-board verification of components by applying the Unit Testing Paradigm and, hence, realizing TDD on reconfigurable hardware. The proposed solution comprises a hardware/software introspection infrastructure to verify modules of a system at different stages, spawning multiple abstraction levels without extra effort nor redesigning the component. Our solution has been implemented for the Xilinx ZynQ FPGA-SoC architecture and applied to the verification of C-kernels within the CHStone Benchmark. Effortless integration into the Xilinx Vivado design flow and tools is supported by a set of automatic generation scripts developed for this end. Experimental results show a considerable speedup of the verification time and unveils inaccuracies concerning the co-simulation estimation obtained by Xilinx tools when compared with the on-board latency measured by our framework.

INDEX TERMS Design for testability, on-board verification, high-level synthesis, FPGA, unit testing framework, test-driven design.

I. INTRODUCTION

Lately, *High-Level Synthesis* (HLS) has broken into the market as the technology that not only provides the ability to speed up FPGA-based designs, but also to ease the burden of verification processes [1]. In turn, HLS expands the horizons of FPGA market since it allows the rapid evaluation of architectural alternatives, regardless of the engineer's experience and thus, makes FPGA-based solutions suitable for both software and hardware engineers [2]. However, HLS lacks of the means to easily verify HLS-based designs across abstraction levels, usually covering only the first steps of the development flow.

Thus, at the highest abstraction level, functional verification of HLS models is performed by writing a testbench using the same high-level language as the one used to specify the hardware component.

This testbench feeds the model of the hardware component with the input vectors (stimuli) and compares the output obtained against a reference (the outcome of the simulation of the so called *golden model*). Later, most of the HLS tools perform the co-simulation of the synthesized, closer-to-hardware RTL (Register-Transfer Level) model, reusing the same testbench. Since the testbench (application/system level) and the RTL model (RTL/Logic level) targets different abstraction levels, different domain languages (i.e. C/C++/SystemC and VHDL/Verilog) are used. Hence, it is necessary to bridge both domains by means of a dedicated software infrastructure. However, the use of system-level models and co-simulation is not the answer to all the challenges posed by the verification of FPGA-based designs. For example, the outcome of the RTL synthesis process is certainly complex to trace back to

The associate editor coordinating the review of this manuscript and approving it for publication was Christian Pilato.

the HLS specification that originated it. Therefore, engineers lose the control over the result, which brings into play additional issues. Also, the accuracy of the verification process is affected when engineers move from one abstraction level to the other.

Furthermore, latest studies [2]–[4] affirm that, for almost one fourth of the projects surveyed, the tasks devoted to ensure the correctness of designs combining an FPGA with other processing technologies account for nearly half of the effort in average, with peaks between 70%-80%. Hence, verification activities are becoming a major concern in FPGA-based projects [4]. Moreover, the validation of HLS-generated modules introduces new demands in addition to checking the mere correctness of their functional behavior. For example, timing checking, which is not usually present in software testing frameworks, is mandatory for HLS-based designs. Also, hardware solutions must be compliant with strict protocol checking rules or meet deadlines in hard real-time systems. Unfortunately, these properties of a design are only exposed at the lowest abstraction levels, and need of the simulation of cycle-accurate models such as RTL specifications. However, RTL simulations are lengthy and proved to be impractical as the size and complexity of the designs grow. Contrarily, on-board verification is fast and accurate, which makes this method the most sensible workaround provided that the designer is given the tools and methods to easy the path down to the FPGA fabric.

The adoption of HLS technology unlocks the possibility to introduce mature, well-known practices from the software realm that could contribute to improve the verification process of custom hardware components, whilst integrating such process into on-board testing flows. In this sense, the testing framework presented in this work, *RC-Unity*, leverages the know-how on *unit testing environments*, which are widely used in software engineering projects, so as to realize the Test-Driven Development (TDD) paradigm on FPGA-based devices. TDD is a software development process that puts the design of the test cases before the actual implementation of the software. This way, software requirements are converted to specific evaluations of the functionality. TDD has been demonstrated to provide a variety of different benefits, such as better productivity [5], better quality [6] and higher test coverage [7].

In this work, an on-board verification framework, based on the principles of TDD, for HLS-based designs is proposed as a response to the current challenges concerning the verification of systems implemented using FPGA technology. The main contributions of this proposal are:

- A hardware verification environment (*RC-Unity*) as an extension of *Unity* software testing framework, that enables the use of a single test suite through the whole hardware design flow, regardless of the abstraction level and the design stage.
- Making the principles of TDD a reality and applying them to the design of FPGA-powered solutions, enabling the development of complex systems in short and safe steps, by verifying the HLS modules these systems are made of.
- A configurable and standardized heterogeneous testing environment that helps engineers to reduce the verification effort, contributing to ease and speed up the path down to the FPGA implementation.

This paper is structured as follows. section II summarizes the current state-of-the-art in the field of on-board verification. Then, in section III the basics of HLS design is briefly introduced. section IV introduces the RC-Unity workflow to apply TDD for reconfigurable hardware. Later, section V shows in detail how RC-Unity can be adopted to build HLS-based modules with special emphasis on its architecture and characteristics, whilst section VI analyzes the impact of this technology applied to the embedded reconfigurable systems. Finally, section VII highlights the main contributions of this work.

II. RELATED WORK

In this section, the focus of the analysis of previous work is put on proposals dealing with the verification of hardware components using FPGA devices, rather than putting the focus on development methodologies. Since the main contribution of *RC-Unity* is the provision of a verification infrastructure to support TDD for FPGA developments, it is sensible to study how verification activities across abstraction levels is approached by other researches, overcoming the lack of on-board verification features of most HLS tools.

Most of the proposed solutions focus solely on the development of the FPGA-based testing platform, where the DUT (Design Under Test) will be deployed, and the communication infrastructure with the host of the tester [8]–[10]. Therefore, these solutions are highly coupled to the architecture and tools, which make them difficult to be reused in other environments that target a different FPGA technology. Feng *et al.* propose in [11] the use of emulation platforms that provide an extra degree of flexibility. These emulation platforms support the acceleration of the DUT, integrated with a model of the rest of the system, at different concretion levels (e.g. functional bus model versus cycle-accurate bus model).

RC-Unity, while providing the same functionality of these verification platforms, effectively abstracts and standardizes the interface to the FPGA by means of the utilization of technology-independent protocols and well-known architectural solutions (design patterns), ensuring its portability across different FPGA families.

On top of the specificity of the majority of the proposals, there is a lack of support for a true, global verification methodology since the reviewed works either overlook this facet or keep the operation of the verification platform at the lowest level of abstraction. That is, feed the DUT, retrieve the output and check for differences. Just a few works go beyond this elementary concept of a testing framework, extending its functionality and the set of abstraction levels covered. To this end, it is used, for example, the semantics and high-level artifacts of System Verilog, SystemC [12] or UVM (Universal Verification Methodology) [13].

UVM establishes a verification framework with a clear separation between the generator of the input test vectors (stimuli) and the verification environment, that is composed by a variety of artifacts, whose aim is to drive the verification process. This separation of roles allows, for instance, the co-simulation of designs at different levels of abstraction since the interface between the DUT and the rest of the testing infrastructure remains stable. Despite the complexity and lengthy set-up time of a fully-compliant UVM environment (one of the principal criticism to this approach) and the dependency on the expertise of the verification engineer (who has to be skilled in this technology), some works such as [14], [15] propose in-hardware verification environments inspired on the principles of UVM. Podivinski *et al.* introduce in [14] the automated generation of the verification environment, an interesting feature which is also supported by *RC-Unity*.

In order to ease the burden of a solution based on UVM for FPGA designs, some works such as [16], [17] foster the reutilization of as many of the components of the platform and the standardization of the verification environment to reduce the developers' effort. Reutilization of C tests through the whole UVM verification flow, at any abstraction level, is proposed by Edelman *et al.* [17] as a way to avoid ''hard to understand'' artifacts (e.g. randomized sequences) for inexperienced UVM practitioners. *RC-Unity* also advocates the principle of reusability as a means to pave the way to decrease the FPGA verification complexity and shorten the verification cycle.

Regardless the different approaches followed by the above mentioned works, the DUT is the minimum introspection level possible during the verification process. Therefore, it is common to see the DUT as a black box. This constraint leads to a limited visibility of what is happening inside the component. If everything runs smoothly, there is no need to see further. However, in the case of a mismatch between the output and the expected result, engineers are hands-tied and no tools are provided to amend this situation. In order to increase the visibility of the DUT internals and provide engineers with better means to enhance the debugging capabilities, Curreri *et al.* present in [18] an HLS technique to efficiently support in-circuit assertions. In this sense, *RC-Unity* provides the ability to check internal blocks/functions of a DUT.

III. FPGA DEVELOPMENT FLOW WITH HLS

HLS technology supports a Specify-Evaluate-Refine (SER) design methodology, targeting the development of accelerators or application-specific processors. Through automation, HLS-based development tools ease the transition from high-level models, describing the behavior of the component, to low-level or implementation models.

The engineer only deals with a specification of the functionality written using a High-Level Language (HLL) such as C or C++. This specification is tuned and optimized until the design requirements are met after several iterations. To this end, the engineer makes it use of the simulation data and resource usage and latency estimation reports produced by the tool.

The verification of the design takes place at three points during the design flow:

A. PROCESSOR/ACCELERATOR LEVEL: SOFTWARE SIMULATION

Along with the model of the component (i.e. the functionality to be accelerated and deployed in the FPGA), a functional verification must be performed and, hence, a set of tests are defined for such purpose. At this level, the verification takes place in the software domain and is focused on the correctness of the behavior, rather than the fulfillment of the timing or resource requirements. This way, engineers get good support that help them to check the exactness of the output generated by the model of the component, following a known set of input stimuli. The reference data set is usually generated through the simulation of an executable reference model also called *golden model*. Software simulations are fast, but they cannot provide information about specific hardware domain aspects such as timing.

B. RTL/LOGIC LEVEL: CO-SIMULATION

Next, the verification of the RTL model is performed through co-simulation. HLS-based development frameworks usually provide the necessary support to allow mixed simulations in order to check for the correctness of the generated code. To this end, the description of the component at logic level (VHDL/Verilog) is instrumented with an adapter that is also generated in an automatic way. This adapter is written in a System-Level Design Language such as SystemC, with capabilities to model both hardware and software, becoming a bridge between domains. Thus, the RTL model can be exercised with the same set of input stimuli provided by the software test bench developed in the previous step, and the outputs can be read in the same way. However, RTL simulations are slow though they provide accurate estimations regarding the latency and throughput of the solution.

C. FPGA/CIRCUIT LEVEL: ON-BOARD VERIFICATION

Finally, the RTL model is synthesized and the output is integrated with the rest of the components in the system. The synthesis process is performed by the tools provided by the vendor, and depends on the particular FPGA technology. The result is a programming file called *bitstream* that is used to configure the FPGA device. Unlike the previous two steps, there is no direct way to bridge the test bench and the hardware implementation in the FPGA device.

D. VERIFICATION CHALLENGES OF HLS-BASED DESIGNS

The previously introduced HLS design flow rises several concerns regarding the verification of heterogeneous embedded systems using FPGA devices.

Nowadays, HLS tools provide an estimation of hardware-specific timing parameters in early stages of the

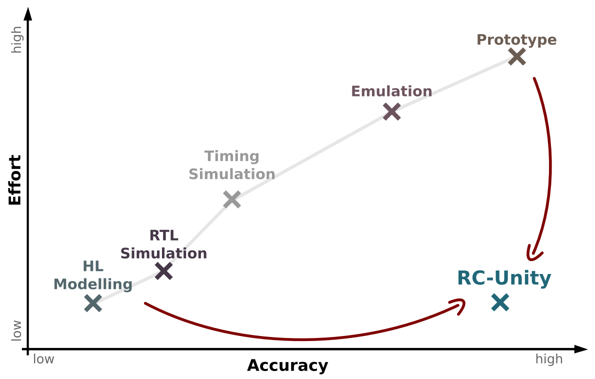


FIGURE 1. **Trade-off between accuracy and simulation effort.**

design flow. However, the accuracy of the results is low and of little help for engineers who should approve the model of the component upon the imprecise information provided by the tool. The functional verification of the system at behavioral level (HLS model in C or C++, for example) is performed using a software model that lacks of any reference to the time domain. On the contrary, at RTL/logic and FPGA/circuit levels engineers can get reliable, cycle-accurate information regarding the behavior of the design intent. Also, on-board verification has an advantage over RTL-based verification that goes beyond the quantitative plane; some hardware bugs are not exposed during simulations and can only be detected once the implemented design is deployed and tested in a real environment/device. Therefore, on-board verification is faster and more precise.

However, verification process using hardware prototypes demand an extra effort due to the need of a testing infrastructure which must be developed specifically for each design, hardly reusable from one project to other (*verification reusability challenge*). This work aims to ease the burden of the verification activities at on-board level; the only level not supported by current HLS tools and design flows such as Vivado HLS suite from Xilinx, enabling the highest accuracy while keeping to a minimum the human intervention (see Figure 2).

It is worth to highlight a second defiance associated with the nature of the established co-design flows for the development of FPGA components using HLS technology. A design model undergoes a series of transformations and refinements until the intent is implemented at circuit level. Each step in this process is devoted to capture different characteristics of the system and makes it use of a complex mixture of languages and models of computation for such end (*verification correctness challenge*). Even though current HLS development frameworks automate the generation of code to move from one abstraction level to other, such code must be verified at each step due to the absence of formal methods that ensure the correctness of such transformations (*verification efficiency challenge*) [19].

IV. TOWARDS TDD FOR RECONFIGURABLE HARDWARE

The proposed verification framework, *RC-Unity*, allows both time and functional introspection of hardware components,

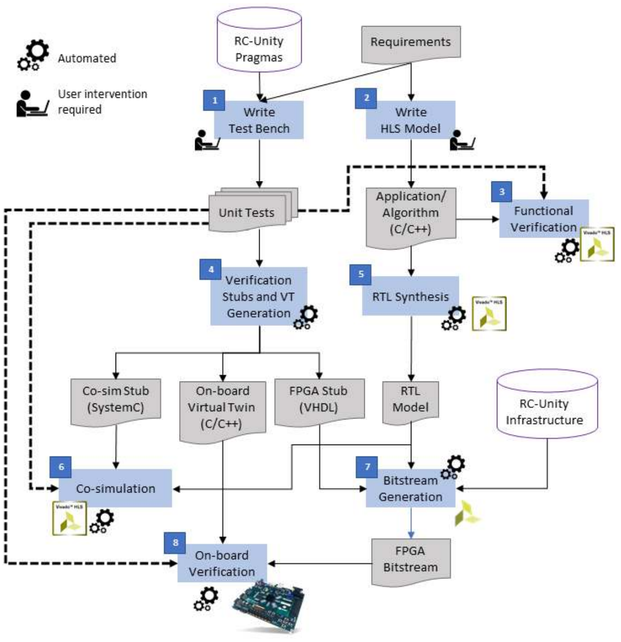


FIGURE 2. ***RC-Unity* workflow.**

modeled and synthesized by means of HLS tools. The key feature of *RC-Unity* is the automatic (i.e., no additional action is required from the verification team) generation of the HW/SW infrastructure. Therefore, there is no need to write a single additional line of code or develop extra hardware infrastructure when the verification team moves down to the board level. On top of that, the test suite intended to check the correctness of the functional model can be used to validate the final implementation on the FPGA.

This fact enables the use of *RC-Unity* to apply Test-Driven Development (TDD), a well-accepted practice in software industry, to the design of embedded systems. Embedded system design frequently implies the development of custom processors or accelerators (using, for example, a reconfigurable logic fabric) that will be later integrated in the final platform. Hence, it is interesting to explore the utilization of TDD during this process and benefit from the gain in productivity [5], quality [6] and high test coverage [7] as it has been proven in the software realm.

Figure 2 illustrates the workflow to be followed by the user of RC-Unity, identifying the manual (user intervention) and automatic steps, together with the principal inputs and outputs to/from the individual processes. The proposed workflow is high-level and generic enough to be easily retargeted to different vendor-specific HLS toolchains as long as they provide basic functionality such as RTL synthesis, co-simulation and FPGA bitstream generation. Only the scripts that automate the generation of the RC-unity stubs and virtual twins (step four), and the board-specific project (step eight) must be revisited. Also, minor changes should be required by the *RC-Unity Infrastructure* components in order to be tailored to the particular features of the target FPGA technology (e.g. HW/SW interfaces, vendor-specific FPGA architectural elements, etc.). In this work, the RC-Unity verification framework has been implemented for the Xilinx ZynQ SoC-7000 architecture and, therefore, the automation scripts and components have been developed to suit Xilinx technology and Vivado toolchain.

Firstly, and as it is established by the TDD paradigm, the *Unit Tests* must be defined. Unit Tests are key in RC-Unity design flow since they are the input to the three verification processes corresponding to the three verification checkpoints established during the development flow. As it can be seen in the picture, the Unit Tests directly feed (dotted line in Figure 2) *Functional*, *Co-simulation* and *On-board* steps, without modifications or adaptations. This is the major advantage of using RC-Unity, developers do not have to revisit the definition of the tests or hand-write custom software and hardware infrastructure to make the Unit Tests compatible with the languages and tools necessary to perform the verification at an specific abstraction level during the design process. In this work, we rely on *Unity* [20] testing framework to guide this task. An extension of the base set of Unity *pragmas* (RC-Unity pragmas) is proposed with the aim to check timing requirements.

Therefore, TDD fosters the use of unit tests, which are the main core of the development process. Unit tests have several characteristics that must follow a designer to write a robust testbench. These attributes are known as F.I.R.S.T. principles [21], which are also met by the unit tests written with RC-Unity.

- *Fast*: Running tests fast enough to not be a practical problem for developers.
- *Independent* or *Isolated*: To avoid any dependence on other tests. One test does not set up the next test.
- *Repeatable*: The test must be repeatable. It must return the same result when it is run in loop.
- *Self-validation*: Test must return a Boolean result (pass or fail), without subjective considerations detecting if passed or not.
- *Timely*: Unit tests should be written just before the production code, preventing bugs.

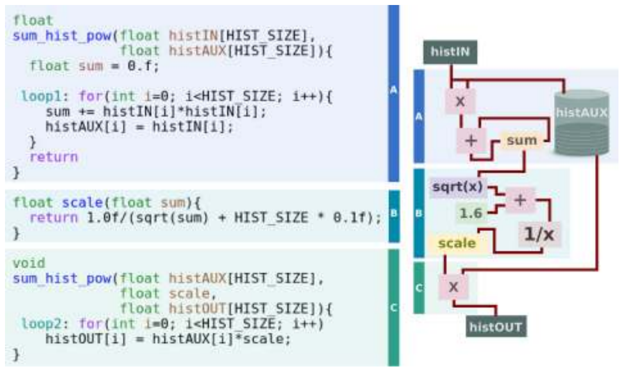
Listing 1 shows an example of how the unit tests look like for a case study; the development of an accelerated version of the HOG (Histogram of Oriented Gradients) feature extractor [22]. The HOG algorithm is widely used in computer vision and image processing for object detection, particularly suited for human detection. The algorithm performs the operation in different steps, being the subject of the unit tests the verification of the *vector normalization* phase (l^2-*norm* function). Each unit test in Listing 1 aims to check for the correctness of one of the three tasks that the l^2-*norm* function comprises, namely: (1) sum of the square root of each pixel; (2) calculation of the scaling factor; and (3) scale of the pixels. Unit test code is clean and easy to write, by means of a set of assertion macros. It is worth introduce the *RC-Unity Time Extensions* (see subsection V-A) which allow to test whether a specific functionality executes in a delimited number of cycles. In this example, the l^2-*norm* algorithm must be completed within 487 cycles: no more than 200 cycles for the first function (line 10 of Listing 1), between 30 and 40 cycles for the second (lines 18 and 19) and no more than 250 cycles for the last one (line 35).

```
 1 #include <rc-unity.h>
 2
 3   void test_sum_hist_pow(){
 4     float input[HIST_SIZE];
 5     for(int i=0; i != HIST_SIZE; i++)
 6       input[i] = i;
 7     TEST_ASSERT_EQUAL_FLOAT(1240.0, sum_hist_pow(input
          ));
 8
 9   #ifdef ON_BOARD
10     TEST_ASSERT_TIME_LT(200);
11   #endif
12   }
13
14   void test_scale(){
15     TEST_ASSERT_EQUAL_FLOAT(0.217391, scale(9.0));
16
17   #ifdef ON_BOARD
18     TEST_ASSERT_TIME_LT(40);
19     TEST_ASSERT_TIME_GT(30);
20   #endif
21   }
22
23   void test_mult_scale(){
24     float ref[HIST_SIZE], input[HIST_SIZE], out[
          HIST_SIZE];
25     for(int i=0; i!= HIST_SIZE; i++)
26       input[i] = i;
27       ref[i] = input[i]*0.1;
28
29     mult_scale(input, 0.1, out);
30
31     for(int i = 0; i != HIST_SIZE; i++)
32       TEST_ASSERT_EQUAL_FLOAT(ref[i], out[i]);
33
34   #ifdef ON_BOARD
35     TEST_ASSERT_TIME_LT(250);
36   #endif
37   }
```

Listing. 1. Test cases for l^2-*norm* algorithm.

Once the unit tests are defined, the implementation of the HLS models (step 2 in Figure 2) for the three components (i.e. *sum_hist_pow*, *scale* and *mult_scale*) must be developed (see Figure 3) using the syntax and semantics of the chosen HLS framework. After this step, *Functional Verification* of the design can be completed. If no errors are detected, verification at RTL level must be accomplished. To this end, *RTL Synthesis* and the generation of the *Co-sim Stubs* must be done beforehand (steps 4 and 5, respectively). The former is performed by the vendor tools, whilst the latter is carried out by the RC-Unity tool *c2UTAdapters*. For each function under test, *c2UTAdapters* tool generates the SystemC stub that bridges the unit test (written in C) and the application/algorithm RTL model (VHDL or Verilog). These stubs are in charge of interpreting the stimuli generated by the unit tests, exercising the *Design Under Test* (DUT), and the outputs from the RTL model. Again, no action from the developer is needed.

Listing 2 shows the report generated by RC-Unity after the verification at functional (simulation) and logic (co-simulation) abstraction levels. In both cases, it is only displayed information about the final status of the tests (i.e., PASS or FAIL).



```
float
sum_hist_pow(float histIN[HIST_SIZE],
             float histAUX[HIST_SIZE]){
  float sum = 0.f;

 loop1: for(int i=0; i<HIST_SIZE; i++){
    sum += histIN[i]*histIN[i];
    histAUX[i] = histIN[i];
  }
  return
}
```

```
float scale(float sum){
  return 1.0f/(sqrt(sum) + HIST_SIZE * 0.1f);
}
```

```
void
sum_hist_pow(float histAUX[HIST_SIZE],
             float scale,
             float histOUT[HIST_SIZE]){
 loop2: for(int i=0; i<HIST_SIZE; i++)
    histOUT[i] = histAUX[i]*scale;
}
```

FIGURE 3. HLS models of the l^2-*norm* steps.

```
ut_l2norm.cc:9:test_sum_hist_pow:PASS
ut_l2norm.cc:20:test_scale:PASS
ut_l2norm.cc:29:test_mult_scale:PASS

-----------------------
3 Tests 0 Failures 0 Ignored
OK
```

Listing. 2. Report for l^2-*norm* functional and RTL verification checkpoints.

After a successful *Co-simulation* (step 6 in Figure 2) of the RTL model, the component is ready to be deployed on the prototyping platform and undergoes the on-board verification process. The component is instrumented in order to be integrated in the RC-Unity verification framework at implementation level. Some of the elements are generic, and are available through a library, and some are specific to the test bed and are automatically generated by the before-mentioned *c2UTAdapters* tool. Thus, the generation of the FPGA bitstream (step 7 in Figure 2) takes these three inputs (i.e., RTL model, FPGA stub and RC-Unity instrumentation components) and produces the hardware side of the on-board test. The software side comprises the same *Unit Tests* written at the beginning of the workflow and the *On-board Virtual Twins* functions. Such virtual twins stand for the actual implementation of the component in the FPGA, exposing an identical functional interface to the *Unit Tests*. In this sense, it ensures the reusability of the test bench up to the physical implementation.

The result of the *On-board verification* (step 8 in Figure 2) is another report with detailed information about execution times. Listing 3 shows an example, displaying not only information about the correctness of the output but the compliance with the timing requirements after the execution of the test. In this example, for *test_scale* and *test_mult_scale* the values of the latency, which are measured directly in hardware by *RC-Unity*, proved to be within the specified ranges, but for *test_sum_hist_pow* the actual execution time exceeds the current limit.

As the reader can notice, the information is displayed in a consistent style, independent of the design phase. Moreover, designers do not need to check manually, whether the test

```
#ut_l2norm.cc:9:test_sum_hist_pow:PASS: Time:FAIL:
     Expected 146 Was 189
#ut_l2norm.cc:20:test_scale:PASS: Time:PASS:Expected
     26 Was 25
#ut_l2norm.cc:29:test_mult_scale:PASS: Time:PASS:
     Expected 81 Was 79

-----------------------
3 Tests 1 Failures 0 Ignored
FAIL
```

Listing. 3. Report of l^2-*norm* with errors (on-board verification).

passed or not, so the Self-Validating F.I.R.**S**.T. principle is fulfilled.

This section sketched the workflow supported by our RC-Unity framework inspired by the TDD philosophy, in which the design of the tests to be passed are the core of the process. This way, and thanks to the automatic generation tools and infrastructure components provided by the proposed solution, it can be faced the development of complex systems in small, fast and safe steps. No extra effort is asked from hardware architects when it comes to the verification of the design at board level. On top of that, cycle-accurate results are obtained easily which allow the improvement of the optimization loops since decisions are made based on actual timing measurements.

V. RC-UNITY: A HARDWARE TESTING FRAMEWORK

As an answer to the before-identified restrictions and challenges concerning the verification of FPGA-based projects, we propose an extension of *Unity* [20] software testing framework for embedded systems that rely on reconfigurable logic devices. *RC-Unity* supports unit testing of hardware accelerators in an easy, straightforward way, based on two main pillars: (1) the utilization of the same test bench regardless the development stage and; (2) the automation of the on-board verification path downwards the prototyping platform.

To achieve this twofold objective, it was necessary to enrich both the syntax and semantics of *Unity* to incorporate concepts typical from the embedded systems such as the verification of timing restrictions. Also, an standardized hardware and software verification infrastructure is proposed to isolate the specific model of the component under test from the testing environment. Thus, the establishment of a predefined set of mechanisms and communication protocols enables the automatic generation of the necessary bridging artifacts across abstraction domains (i.e. functional, RTL and, implementation). Next, we provide the reader with an insight of the main concepts of our *RC-Unity* proposal.

A. RC-UNITY TIME EXTENSION

To enable time analysis, *Unity* framework [20] has been extended with new features that allow: (1) the unit test to interact with the software and hardware components of the verification infrastructure and; (2) to report to the developer information related to the execution time of the test cases. Table 1 lists the set of macros available in *RC-Unity*.

TABLE 1. ***RC-Unity* macros.**

Macro	Description
`RCUNITY_RESET`	Sets the verification environment to a well-known state
`RCUNITY_CONF_TIME(time)`	Configures the time budget for an operation. By default 0 (it means no limit of time)
`RCUNITY_START`	Enables the component under test during the time depicted in `RCUNITY_CONF_TIME` macro
`RCUNITY_CONF_SKIP_INPUT(num)`	Modifies the number of clock cycles to delay the beginning of the internal counter for an operation. By default 1
`RCUNITY_CONF_SKIP_OUTPUT(num)`	Ignores the last *num* clock cycles of an operation so that trailing computations are left out. By default 1
`TEST_ASSERT_TIME_XX(expected)`	Compares the time obtained and expected value in accordance with the comparison operator, where `XX` is: EQ (equal than), LT (less than), GT (greater than), LE (less or equal than) or GE (greater or equal than)

These extensions have been written in ANSI C, which makes it fairly portable across unalike platforms such as 8-bit microcontrollers or 64-bit processors, taking advantage of the *Unity* capability to extend its functionality and adapt the framework to the particular requirements of the project.

Briefly, it is detailed the role of each macro. RCUNITY_RESET macro sets the testing environment to a well-known initial state and RCUNITY_CONF_TIMER configures a timer that establishes the operation time of the component under test. RCUNITY_START activates the component, starting an internal counter and waiting for the component to complete its operation, unless a limit of time had been established using RCUNITY_CONF_TIMER. By default, it is measured the time for an operation taking as the reference the first read operation (start time) and the first write operation (stop time) issued by the component. This behaviour can be modified if necessary; the RCUNITY_CONF_SKIP_INPUT and RCUNITY_SKIP_OUTPUT macros configures the moment the *begin* and *end* events are captured to start/stop the counter. Last but not least, the macro TEST_ASSERT_TIME compares the elapsed time (retrieved from the platform) against the value of the macro parameter which expresses clock cycles. Several comparison operators are available in RC-Unity such as EQ (equal than), LT (less than), GT (greater than), LE (less or equal than) and GE (greater or equal than). Configuration macros are *lazy* operations, they take effect after the RCUNITY_START macro is executed.

Listing 4 shows the C implementation of the `RCUNITY_RE SET` macro for an embedded OS based on GNU/Linux. Such macro operates the *Test Manager* component (see subsection V-B), which is abstracted by the operating system as a memory-mapped peripheral that can be accessed from software through the use of the POSIX-compliant *mmap* primitive (lines 6-16 of Listing 4). Once a pointer to the memory region, where the *Test Manager* peripheral is mapped, is obtained the software writes (line 18 of Listing 4) `ID_FUNC_RESET` on register 0 × 00 (`REG_OPERATION`), indicating the requested operation (a reset, in this case).

The rest of the extended macros listed in Table 1 follow the same pattern as the `RCUNITY_RESET` macro and those macros that contain parameters include some writings after the identifier word, following the same operation that shows line 18 of Listing 4. Contrary, the `TEST_ASSERT_TIME_XX` macros do a reading instead of a writing operation to get the time elapsed between the start and the end of the operations done by the hardware component that is being verified.

```
 1  #define RCUNITY_RESET _RCUnityReset()
 2
 3  void
 4  _RCUnityReset()
 5  {
 6    void *ptr;
 7    unsigned page_size=sysconf(_SC_PAGESIZE);
 8
 9    int fd = open ("/dev/mem", O_RDWR);
10    if(fd < 1)
11    {
12      printf("Cannot open /dev/mem for writing\n");
13      return;
14    }
15    ptr = mmap(NULL, page_size, PROT_READ|PROT_WRITE,
16    MAP_SHARED, fd, REG_OPERATION);
17
18    *((unsigned *)ptr) = ID_FUNC_RESET;
19    munmap(ptr, page_size);
20  }
```

Listing. 4. **C implementation of *RCUNITY_RESET* macro.**

```
 1  #include <unity.h>
 2  #include <factorial.h>
 3
 4  void
 5  test_Natural()
 6  {
 7    #ifdef ON_BOARD
 8      RCUNITY_RESET
 9      RCUNITY_START
10    #endif
11    TEST_ASSERT_EQUAL_INT(720, factorial(6));
12    #ifdef ON_BOARD
13      TEST_ASSERT_TIME_LT(50);
14    #endif
15  }
```

Listing. 5. **Example of test case (*factorial*) using RC-Unity.**

In order to illustrate how easy and straight forward is the writing process of unit tests in *RC-Unity*, a test case for a hardware component implementing the *factorial* function is shown in Listing 5. In this example, the goal of the test case is twofold: on the one hand, to check that the result produced by the component is correct (line 11 of Listing 5); on the other hand, to assure that the time taken by the *factorial* function

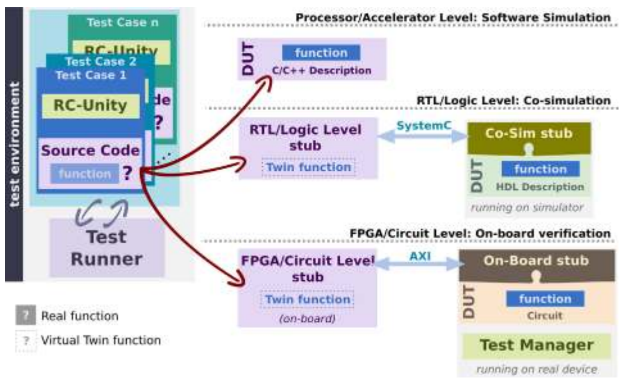
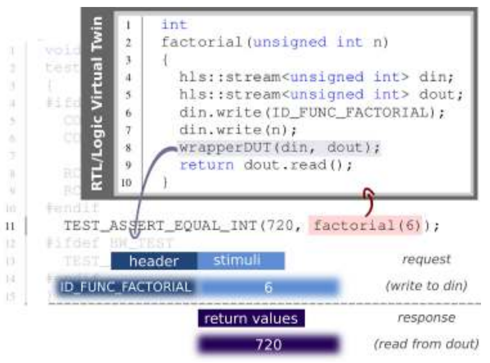


FIGURE 4. RC-Unity verification framework.



```
int
factorial(unsigned int n)
{
  hls::stream<unsigned int> din;
  hls::stream<unsigned int> dout;
  din.write(ID_FUNC_FACTORIAL);
  din.write(n);
  wrapperDUT(din, dout);
  return dout.read();
}
```

```
  TEST_ASSERT_EQUAL_INT(720, factorial(6));
```

FIGURE 5. RTL/Logic level implementation of the virtual twin function (*factorial* function example).

does not exceed a specific number of cycles (line 13 of Listing 5).

When performing the verification of the design at FPGA/Circuit level (on-board verification), the hardware infrastructure needs to be configured beforehand. *RC-Unity* provides a neat way to accomplish this task through the insertion of the necessary *RC-Unity extension macros* (lines 7-10 of Listing 5). By using these extensions, the test case becomes fully reusable, without any changes, through the whole design process, and makes it Repeatable and Independent (F.**I**.**R**.S.T. principles) of other test cases; both principles are fulfilled because the reset *pragma* sets the DUT in a well-known state, removing the configuration and/or data stored in internal memories of previous test cases.

B. HW/SW VERIFICATION INFRASTRUCTURE

In order to add value to standard HLS-based design and verification flows, *RC-Unity* specifies a testing infrastructure that allows the reutilization of the same test suite throughout the entire lifecycle of a HLS project.

Figure 4 represents the main elements of the proposed verification framework. The different elements of the framework are either pre-defined, customized (depending on the stage of the design flow) components available to the engineer or generated in an automatic way.

The verification framework comprises the following software entities: (1) the set of *Unit Tests* that make it use of the RC-Unity extension macros (see subsection V-A); (2) the *Test Runner* which calls upon each unit test case sequentially; (3) the co-simulation and FPGA virtual twins (see subsection V-C. On the hardware side, the FPGA bitstream includes the component to be verified surrounded by a tailored, automatically generated *on-board stub* and an instance of the *Test Manager*, from the RC-Unity hardware infrastructure library.

Next, we proceed to explain the principal features of the above mentioned components and their role during the different steps of the verification process.

To execute the unit tests, a *Test Runner* is automatically generated using the Ruby script provided by *Unity* framework. The Test Runner is a C file that contains the *main* function that launches the verification process. The code of the Test Runner does not change, regardless the abstraction level of the specification model of the design. To begin the execution of a test, the Test Runner makes a call to the top level function of the test. For example, *test_Natural* in Listing 5. The actual implementation of the so called *Design Under Test* (DUT) depends on the step we are in the development process. Thus, we may have a functional software model of the DUT, an RTL model and a bitstream physically deployed on the FPGA. In order to make the Test Runner independent on the implementation details of the DUT, it does not interact directly with the DUT. The only exception is when the functional verification takes place since both (Test Runner and DUT) are defined using C programming language. By contrast, for co-simulation and on-board verification, the Test Runner interacts with the DUT through the RTL and FPGA *virtual twins*, respectively.

A *virtual twin* is C/C++ code generated in an automatic way that stands for the actual implementation of the DUT. It exposes exactly the same interface as its functional counterpart (see implementation of the *factorial* virtual twins in Figures 5 and 7), so the Test Runner is detached from the abstraction level, language or technology employed to model or implement the DUT at any given design step (see Figure 4).

The *virtual twin* knows how to trigger the execution of the actual DUT by using the services and API provided by the platform. Invocations from the Test Runner (C or C++ software calls) are conveniently translated and redirected, depending on the verification stage. Therefore, as it can been seen in Figure 4, the *virtual twin* at RTL/Logic level exercises the VHDL/Verilog DUT logic model using the SystemC instrumental code generated by Vivado HLS tool. Nevertheless, the *virtual twin* at FPGA/Circuit level converts the DUT invocations into AXI bus transactions at hardware level.

The *Test Manager* is a hardware component that intermediates between the Test Runner and the FPGA implementation of the DUT. It actually implements: (1) the time-related functions listed in Table 1; and (2) the functionality related with the configuration, execution and control of the *hardware verification environment*. By identifying and embedding this functionality into an independent component, *RC-Unity* decouples the testing environment from the DUT. This fact facilitates the portability of the testing framework to scenarios involving different languages and technologies.

The Test Manager exposes an standardized interface and protocol to both the FPGA virtual twin and the FPGA implementation of the DUT. The FPGA virtual twin translates either commands (*RC-Unity* macros) or data (stimuli) to AXI bus transactions that are received by the Test Manager. Then, the Test Manager interprets such transactions and proceed accordingly.

The *DUT* is instrumented (on-board stub) to provide it with the capability to interact with the Test Manager (the real interface to the rest of the verification environment). A custom, automatically generated VHDL wrapper implements the necessary functionality: (1) interpretation of the commands issued by the *Test Manager*; (2) gathering of the DUT execution statistics. The on-board stub also interfaces with the DUT at signal level in order to control (i.e. start, stop) and monitor its operation.

C. INSTRUMENTATION: STUBS AND VIRTUAL TWINS

RC-Unity has been specifically conceived to help in the verification tasks of systems that make it use of reconfigurable technology to implement hardware accelerators. Nowadays, leading FPGA manufacturers deliver embedded platforms that combines a reconfigurable logic fabric with standard hard processors (FPGA-SoCs), also known as Programmable Logic (PL) and Processing System (PS), respectively. Such hybrid devices exploit the diversity offered by multiple processing technologies, allocating computing tasks to the cores whose technology better fits the requirements of the target application (e.g. high-performance, low-power, etc.).

In our approach, the unit tests are always executed on the PS side, whilst the DUT will be mapped on the PL. Such mapping leads to certain interfacing challenges between HW-SW artifacts that are solved by means of an RMI (Remote Method Invocation) approach [23]. In this sense, *RC-Unity* offer a cross-domain solution that bridges the test environment and the DUT in a transparent way, despite the fact the latter might be modeled or implemented using different specification languages depending on the abstraction level. As it was mentioned before, *RC-Unity* delegates this responsibility to the *stubs* and *virtual twins*, which both are automatically generated.

The generation process of these instrumentation elements is accomplished by means of the application of the RMI paradigm [24]. RMI was initially proposed in networked distributed systems as a mechanism to allow interaction between components deployed physically distant. The main advantage of RMI is that it provides orthogonality between functionality and communication; any method invocation must take place between special components that provide the endpoints of the communication with the illusion that they are interacting directly.

This separation of concerns (functionality and communication) is applied in *RC-Unity* so as to achieve effective decoupling between the different components in the testing framework. Each module has a functionality assigned which constitutes its interface. When this functionality is known beforehand, *RC-Unity* implements a static version of the *stubs* and *virtual twins* such as it is the case of the *Test Manager*. But, when the functionality to execute depends on the design, as it is the case of the DUT, *RC-Unity* provides the designer with an interface compiler (the *c2UTAdapters* tool) that automatically generates the *stubs* and *virtual twins* taking as the sole input the functional interface definition of the components.

Figure 5 shows the virtual twin of the *factorial* DUT for co-simulation whereas Figure 7 depicts the virtual twin implementation for on-board verification purposes. The code, generated by the *c2UTAdapters* tool, is ready to be used directly either in a Vivado HLS project or executed by the Processing System of the FPGA-SoC.

The signature of the *factorial* function interface specifies a 32-bit integer as the input and returns another 32-bit integer. As it can be observed, the factorial virtual twin maintains the original signature of the DUT (lines 1 and 2 of Figures 5 and 7) so it can be invoked directly by the unit test, just as it is done during software verification (line 11 of Listing 5).

1) RTL/LOGIC VIRTUAL TWIN

When it comes to co-simulation, Vivado HLS will generate the necessary SystemC glue logic to inject stimuli to the RTL model of the DUT produced during the synthesis process. The RTL/Logic virtual twin prepares the input arguments (marshalling) as they are expected by the co-simulation stub, which decodes (unmarshalling) the data and exercises the actual implementation of the DUT. Therefore, the virtual twin plays the role of the *proxy* and the Co-Sim stub behaves as the *skeleton*, attending to the architecture defined by the before mentioned RMI paradigm.

The DUT HLS model is instrumented through the definition of two channels; the input channel is used to feed the co-simulation stub (`wrapperDUT` function in line 8 of Figure 5) with the arguments serialized following a common packet format, whilst the output channel is used to read the result obtained from the DUT. The result is also serialized, following the same marshalling rules defined by RC-Unity.

Lines 4 and 5 of Figure 5 define the input/output channels using the Vivado HLS class *hls_stream*, and lines 6 and 7 build the RMI message. In the example represented in the figure, the invocation message consists of two 32-bit words. The first one is the function identifier (necessary since the co-simulation stub can be the front end for various DUT) and the second one is the number of stimuli. Then, the actual *factorial* function is executed (line 8 of Figure 5). The marshalling and unmarshalling mechanisms are much more complex than those represented in this example, since the processing of the packet depends on multiple parameters such as the datawidth of the communication channels, alignment, number and type of the input/output arguments, etc. In this case, the interpretation of the result does not need further processing, and the data pop out of the output stream

TABLE 2. Latency statistics and timing accuracy errors of CHStone kernels.

Kernel	Num. Tests	Latency Statistics						Accuracy Errors			
		Co-Simulation			On-board Verification (Our)			Minimum Error		Maximum Error	
		Min[1]	Max[1]	AVG[1]	Min[1]	Max[1]	AVG[1]	Absolute	Relative	Absolute	Relative
adpcm	2	54	348	201	11	339	175	43	390.91%	9	2.59%
aes	2	3002	6712	4857		4526		1524	33.67%	2186	32.57%
blowfish	1	0	29239	14620		29243		4	0.01%	4	0.01%
dfadd	40	8	20	14	8	19	14	0	0.0%	1	5.0%
dfdiv	21	9	317	163	8	160	84	1	12.5%	157	49.53%
dfmul	20	8	23	16	8	19	14	0	0.0%	4	17.39%
dfsin	36	22	6466	3244	21	2166	1094	1	4.76%	4300	66.5%
gsm	1		3647			3646		1	0.03%	1	0.03%
jpeg	1		405962		385030	405963	395497	20932	5.44%	1	0.0%
mips	1		3536			3535		1	0.03%	1	0.03%
motion	1	0	215	108	0	216	108	0	0.0%	1	0.47%
sha	1		127437			127436		1	0.0%	1	0.0%

[1] In clock cycles [2] In number of 32-bit words [3] In Words per clock cycle

```
1  void
2  wrapperDUT(hls::stream<unsigned int> din, hls::stream<
       unsigned int> dout)
3  {
4    getRequestHead(din, header);
5
6    if (ID_FUNC_FACTORIAL == header.methodID){
7      readParameters_FACTORIAL(din, args_FACTORIAL);
8      ret_Factorial._return = factorial(args_FACTORIAL.n
           );
9      buildResponse(dout, ID_FUNC_FACTORIAL,
10             sizeof(ret_Factorial)/sizeof(int));
11   }
12   else{
13     discard(din, header.size);
14     buildResponse(dout, ID_FAIL, 0);
15   }
16 }
```

Listing. 6. Example of RTL/Logic stub (*factorial*).

is directly returned by the factorial RTL/Logic virtual twin to the unit test (line 9 of Figure 5).

2) RTL/LOGIC STUB

The RTL/Logic stub is the main responsible for executing the logic that implements a particular function. To do this, the stub parses the header, which is sent by the test case through the `din` port, and stores the identifier of the function that must be invoked and together with the size of the data it expects to be received in a struct (line 4 of Listing 6). If the identifier does not match with a known function, the stub will discard as many 32-bit words as the indicated by the size field (lines 12 to 15 of Listing 6). On the contrary, if the identifier is known, the stub will call upon the corresponding function (line 8 of Listing 6). Before the function call, the stub extracts the input parameters that are required by the function. In our example, the stub gets the unique parameter needed by the *factorial* function (n) through the `readParameters_Factorial` function, which is automatically generated from the signature of the *factorial* function (see line 7 of Listing 6).

Once the result of the function is ready, the stub builds the response message that is sent through the `dout` port. Such response includes a small header that contains the identifier of the call function and the size of the result expressed in 32-bit words (lines 9 and 10 of Listing 6, respectively).

In this sense, RTL/Logic stub provides the mechanisms to ensure the Timely F.I.R.S.**T**. principle. Unlike software, hardware designs must include a mechanism to discard call invocations that have not been implemented yet. Thus, the function identifier, included in the header, allows to decide whether there is code to manage it (lines 12 to 15 of Listing 6) or not (lines 6 to 11 of Listing 6).

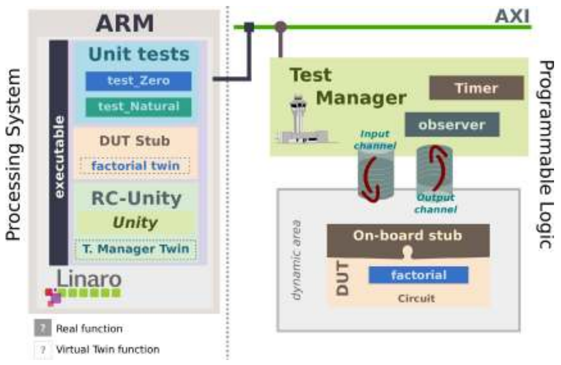


FIGURE. 6. RC-Unity on-board verification environment.

3) ON-BOARD VERIFICATION INFRASTRUCTURE

The last step in the design flow of an FPGA-based accelerator is the implementation of the RTL model. This process, called synthesis, generates the *bitstream*, that is, the programming file of the FPGA. Only then is possible to test the DUT under real conditions; this process is also known as on-board verification. This type of verification has two main advantages: (1) it saves time since hardware execution is faster than RTL simulation; (2) it increases the accuracy of the results (see Table 2 in section VI) and; (3) since it is the final hardware implementation, it may unveil any potential inconsistencies introduced by synthesis tools, as in the case of interface protocols at signal level.

In this work, a prototype of the proposed verification environment has been developed for the Xilinx ZynQ SoC-7000 architecture. Such architecture combines a dual ARM Cortex A9 processor in the PS part and a Xilinx 7-series FPGA in the PL side. In this sense, Figure 6 shows how the different *RC-Unity* entities are mapped on hardware resources in the FPGA System-on-Chip. In this case, the PS is responsible for executing the *RC-Unity* framework and the unit tests on top of a Linaro Ubuntu distribution (a GNU/Linux-based embedded OS), whereas the PL hosts the synthesized

Test Manager and the DUT, together with its communication adapter (on-board stub).

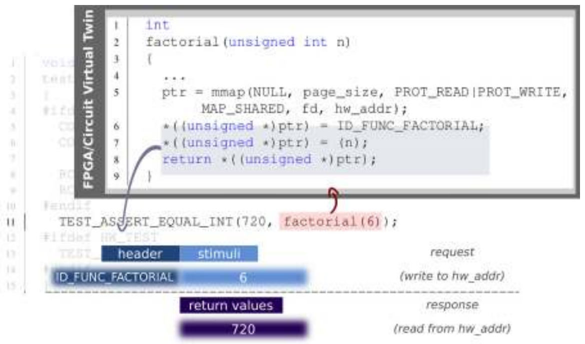
Both, the PL and PS are connected using AMBA (Advanced Microcontroller Bus Architecture) technology that enables the transfer of data between them. Thus, the *Test Manager* component is connected to an AXI bus and receives messages that encapsulate either commands (*RC-Unity* macros) to control and monitor the verification process, or data (stimuli) to exercise the DUT. Although the depth of the internal memory of the *Test Manager* component is customizable at design time, the data transfer process is carried out through an off-chip memory (DDR), in which stimuli and execution traces are stored. In this sense, test cases write the stimuli in a reserved region of the DDR, whilst the *Test Manager* reads such stimuli to exercise the DUT. The output of the DUT can be also stored in the DDR by the *Test Manager*, and then the test case reads the output from the DDR to compare it with the golden values. Thus, the *Test Manager* plays the role of a DMA with additional capabilities, such as measuring the time elapsed to perform a task by the DUT. It is worth mentioning that designers must allocate enough memory space in the DDR for the stimuli and output values.

As in the case of the RTL/Logic stub, the on-board stub is automatically generated by the *c2UTAdapters* tool. Also, the communication between the FPGA virtual twin and the *Test Manager* follows the same RMI principles as the ones described for the communication between the co-simulation virtual twin and the RTL/Logic co-simulation stub.

Figure 7 represents the actual implementation of the FPGA virtual twin for the *factorial* function. It is worth pointing out that this version of the virtual twin maintains the same signature (lines 1 and 2 of Figure 7) than the one generated for the RTL/Logic level (see Figure 5). Therefore, the Test Runner does not have to change its behavior and the unit test has not to be re-written, achieving one the goals set in this work. Thus, the virtual twin implementation for on-board verification purposes maps the input/output channels of the DUT to memory, accessible by the user process (line 5 of Figure 7). In fact, it is the Test Manager who holds a register of the DUT channels mapped to memory, and the one that redirects the traffic to the DUT, behaving as a transparent bridge. This is implemented this way because the on-board Test Manager accounts for specific functionality (i.e. gathering statistics) that requires awareness of the transactions between the testing environment and the DUT. The virtual twin uses the pointer to memory as a read/write stream, just like the interface defined at RTL/Logic level where the *hls::stream* abstraction is used. In both cases, the communication stub builds the RMI message that, in the end, will be translated into transactions over an AXI bus (lines 6-8 of Figure 7).

VI. RC-UNITY IN PRACTICE: VIVADO HLS FRAMEWORK USE CASE

In order to demonstrate the strengths and potential of *RC-Unity*, our verification framework has been tested for a collection of C-kernels defined by the CHStone bench-



```
1  int
2  factorial(unsigned int n)
3  {
4    ...
5    ptr = mmap(NULL, page_size, PROT_READ|PROT_WRITE,
          MAP_SHARED, fd, hw_addr);
6    *((unsigned *)ptr) = ID_FUNC_FACTORIAL;
7    *((unsigned *)ptr) = (n);
8    return *((unsigned *)ptr);
9  }
```

```
11   TEST_ASSERT_EQUAL_INT(720, factorial(6));
```

FIGURE. 7. **FPGA/Circuit level implementation of the virtual twin function (*factorial* function example).**

mark [25]. This benchmark suite comprises a selection of algorithms from various application domains, aiming to set a comparison framework for HLS tools.

The architecture of the proposed framework can be easily adapted to different FPGA-SoC platforms and toolchains, due to its modularity and the use of design patters such as RMI (that isolates the technology-dependent components and enables the automatic generation of the low-level implementation routines). Nevertheless, in this article it has been selected the Vivado HLS tool for evaluation purposes. Free licensing and a complete workflow down to the board level were key in this choice. It is also worth mentioning that the comparison with other commercial or academic HLS tools is, in many cases, complex if not impossible. For example, tools such as LegUp only performs RTL synthesis and, then, it relies on third-party tools to simulate RTL and generate the bitstream out of the VHDL or Verilog code. So, not functional or RTL/Logic co-simulation is supported which handicaps the possibility to establish a fair comparison. In this case, it would be necessary to perform the necessary interface adaptations to make it compatible the semantics of the RTL interface generated by LegUp with the one defined by RC-Unity. This way, the DUT would be compatible with the rest of the infrastructure.

The aim of this section is to show the reader a glimpse of the main capabilities of RC-Unity when it comes to the improvement of the verification process of FPGA-based developments (i.e. reduced verification times and better accuracy) for a specific use case such as Vivado. In future work, RC-Unity support to other tools or HLS frameworks could be extended. It would be only necessary to update the backend of the *c2UTAdapters* to include the implementation details for a specific platform or technology that is realized by the different virtual twins and stubs, whilst the rest of the framework components (e.g. Test Manager or unit tests) would remain unmodified. This would be the case of CatapultC, for example. Since it does have support for functional and mixed simulations but not for on-board verification it would be necessary to rewrite some parts of the backend.

Table 2 shows a comparison of the simulation and execution times for the twelve kernels present in CHStone. For each kernel, a variable number of tests have been conducted

TABLE 3. Latency and timing accuracy errors of `dfsin` kernel.

Test Case	Input Value	Latency statistics*		Accuracy-Error	
		Co-Simulation	On-board (Our)	Relative	Absolute
Test 1	0	22	21	1	4.54%
Test 2	$\pi/18$	392	372	20	5.10%
Test 3	$\pi/9$	942	551	391	41.50%
Test 4	$\pi/6$	1472	551	921	62.56%
Test 5	$2\pi/9$	1832	732	1100	60.04%
Test 6	$5\pi/18$	2007	726	1281	63.82%
Test 7	$\pi/3$	2182	726	1456	66.72%
Test 8	$7\pi/18$	2361	911	1450	61.41%
Test 9	$4\pi/9$	2546	911	1635	64.21%
Test 10	$\pi/2$	2737	917	1820	66.49%
Test 11	$5\pi/9$	2906	1080	1826	62.83%
Test 12	$11\pi/18$	3073	1078	1995	64.92%
Test 13	$2\pi/3$	3236	1080	2156	66.62%
Test 14	$13\pi/18$	3242	1086	2156	66.50%
Test 15	$7\pi/9$	3427	1263	2164	63.14%
Test 16	$5\pi/6$	3614	1267	2347	64.94%
Test 17	$8\pi/9$	3795	1267	2528	66.61%
Test 18	$17\pi/18$	3811	1279	2532	66.43%
Test 19	π	3994	1450	2544	63.69%
Test 20	$19\pi/18$	4171	1444	2727	65.38%
Test 21	$10\pi/9$	4334	1442	2892	66.72%
Test 22	$7\pi/6$	4499	1615	2884	64.10%
Test 23	$11\pi/9$	4670	1615	3055	65.41%
Test 24	$23\pi/18$	4845	1617	3228	66.62%
Test 25	$8\pi/6$	4543	1613	2930	64.49%
Test 26	$25\pi/18$	5028	1800	3228	64.20%
Test 27	$13\pi/9$	5209	1798	3411	65.48%
Test 28	$3\pi/2$	5396	1800	3596	66.64%
Test 29	$14\pi/9$	5400	1804	3596	66.59%
Test 30	$29\pi/18$	5573	1971	3602	64.63%
Test 31	$15\pi/9$	5754	1981	3773	65.57%
Test 32	$31\pi/18$	5921	1971	3950	66.71%
Test 33	$16\pi/9$	5929	1979	3950	66.62%
Test 34	$33\pi/18$	6098	2150	3948	64.74%
Test 35	$17\pi/9$	6293	2166	4127	65.58%
Test 36	$35\pi/18$	6466	2152	4314	66.71%

*In clock cycles

(second column). The simulation time has been obtained by using the co-simulation feature that is present in the Vivado HLS tool. The actual execution time has been measured by means of our *RC-Unity* on-board verification framework.

For each kernel, the average simulation and execution times have been calculated, as well as the minimum and maximum difference observed between co-simulation and on-board testing (represented in absolute and relative terms in Table 2) when comparing to the latency measured in hardware. Comparing both strategies, it can be conclude that the results obtained using co-simulation can lead in certain cases to erroneous design decisions due to the actual difference when measuring the latency on-board using *RC-Unity*.

Noteworthy is the significant error detected for some of the CHStone kernels, which ranges from 5% to 66%. Focusing on the `dfsin` kernel, which is the one with the highest relative error, Table 3 shows that the error increases with the number of test cases. It has been observed that Vivado HLS builds a queue of stimuli, one entry per input to the DUT. The latency of a single execution is measured starting from the time the stimuli for such test case is inserted in the queue not when the processing of the stimuli begin. Therefore, this fact produces wrong timing profiling, leading the developer to wrong conclusions. Our solution measures the real time elapsed just when the stimuli are read by the function that the test case exercises, and hence, it provides more accurate timing results. A special mention has to be made for the results of unit test `adpcm` where the unit test with the minimum latency exposes an error between co-simulation and on-board execution of 390.91%. In this case, we could not conclude that the SystemC bridge generated by Vivado HLS tool was responsible for such difference as in the previous two cases. Further study of the co-simulation infrastructure was not conclusive. From the point of view of RC-Unity's role, it is important to highlight that the proposed framework unveiled such situation, demonstrating that the designer should not rely 100% on the estimations provided by the tools. So, this is a representative case of accuracy improvement using our approach.

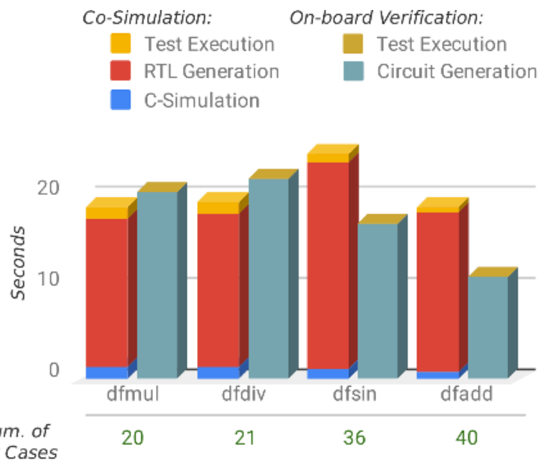


FIGURE. 8. Trade-off between test execution and number of tests.

One of the aims of this work is to reduce as much as possible the FPGA-in-the-loop verification time. This is a key requirement to reduce the wait times and, therefore make it sensible the application of a TDD approach on reconfigurable hardware. In this sense, the generation of the bitstream, the programming file of the FPGA, is a costly task in terms of time and energy, this work fosters the reuse of those parts of the *RC-Unity* FPGA infrastructure that do not change from one verification project to other.

Thus, the DUT is the only part of the project that has to be synthesized, reducing significantly the overall time spent by one iteration of the verification loop. Figure 8 compares the co-simulation (RTL/Logic level) and on-board (FPGA/Circuit level) verification times for the DSP kernels of the CHStone benchmark that have enough test cases to analyze timing trends. The co-simulation time comprises the simulation of the C model (functional verification), RTL generation and test execution, while the on-board verification time includes circuit implementation (RTL generation and synthesis) and test execution. The chart shows that the cost of verification at RTL/Logic level (co-simulation) grows with the number of tests, reaching a point that makes it more costly than verification at FPGA/Circuit level (on-board). It must be recalled that, when a new test case is added, co-simulation must perform the generation of the RTL model again. This process entails the instrumentation of the DUT and the generation of the SystemC stubs, necessary to allow mixed language simulation, every time a new test case is added before the execution. Contrarily, on-board verification only has to execute the test cases. The time to generate the bitstream contributes just once, and it is shared between all the test cases (i.e. the larger the number of tests, the smaller the impact) which makes it a more scalable solution. Thus, the Fast **F**.I.R.S.T. principle is met; test cases are executed as fast as possible, taking a small amount of seconds (see Figure 8), even if there are thousands of unit tests.

VII. CONCLUSION

In this paper, *RC-Unity* verification framework has been introduced. *RC-Unity* is a comprehensive proposal that leverages unit testing of HLS-based hardware modules to ease and improve the design of embedded systems targeting an FPGA fabric. To the best of our knowledge, this is the first work facing the development of an on-board verification environment that is fully reusable during the whole design cycle, regardless the abstraction level of the system specification. Thus, engineers are provided with the tools for generating, in an automatic way, the necessary HW/SW infrastructure, saving time and effort. On top of that, *RC-Unity* enables TDD to embedded system design, making it possible to import the multiple benefits of this popular software development methodology.

RC-Unity provides support to perform both functional and timing verification; the latter is a specific new challenge, derived from the nature of real-time and hardware projects. By means of a technology-independent architecture and the use of well-known design patterns such as *Remote Method Invocation*, *RC-Unity* offers a multi-domain solution while maintaining the test suite through the whole project lifecycle. Thus, the proposed verification framework spans across the three main abstraction levels, as to the detail in the specification of an embedded system (i.e. functional, RTL and implementation/physical).

Finally, automatic generation makes this proposal accessible and easy to use for both hardware and, specially, software engineers who do not need to put extra effort to quickly draw on the benefits of this technology.

REFERENCES

[1] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, ‘‘High-level synthesis for FPGAs: From prototyping to deployment,’’ *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 30, no. 4, pp. 473–491, Apr. 2011.

[2] P. Mishra, R. Morad, A. Ziv, and S. Ray, ‘‘Post-silicon validation in the SoC era: A tutorial introduction,’’ *IEEE Des. Test. Comput.*, vol. 34, no. 3, pp. 68–92, Jun. 2017.

[3] H. Foster, ‘‘2020 Wilson research group functional verification study: FPGA functional verification trend report,’’ Mentor, SIEMENS EDA, Plano, TX, USA, Tech. Rep., 2020. [Online]. Available: https://s3.amazonaws.com/s3.mentor.com/public_documents/whitepaper/resources/mentorpaper_108468.pdf

[4] W. Chen, S. Ray, J. Bhadra, M. Abadir, and L.-C. Wang, ‘‘Challenges and trends in modern SoC design verification,’’ *IEEE Des. Test. Comput.*, vol. 34, no. 5, pp. 7–22, Oct. 2017.

[5] 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, Mar. 2005.

[6] 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. Eng.*, vol. 13, no. 3, pp. 289–302, Jun. 2008.

[7] D. Janzen and H. Saiedian, ‘‘Does test-driven development really improve software design quality?’’ *IEEE Softw.*, vol. 25, no. 2, pp. 77–84, Mar. 2008.

[8] L. De Luna and Z. Zalewski, ‘‘FPGA level in-hardware verification for DO-254 compliance,’’ in *Proc. IEEE/AIAA 30th Digit. Avionics Syst. Conf.*, Oct. 2011, pp. 7D5-1–7D5-5.

[9] X. Cheng, A. W. Ruan, Y. B. Liao, P. Li, and H. C. Huang, ‘‘A run-time RTL debugging methodology for FPGA-based co-simulation,’’ in *Proc. Int. Conf. Commun., Circuits Syst. (ICCCAS)*, Jul. 2010, pp. 891–895.

[10] A. Wicaksana, A. Prost-Boucle, O. Muller, F. Rousseau, and A. Sasongko, ‘‘On-board non-regression test of HLS tools targeting FPGA,’’ in *Proc. 27th Int. Symp. Rapid Syst. Prototyping: Shortening Path Specification Prototype*, Oct. 2016, pp. 1–7.

[11] L. Feng, Z. Dai, W. Li, and J. Cheng, ‘‘Design and application of reusable SoC verification platform,’’ in *Proc. 9th IEEE Int. Conf. ASIC*, Oct. 2011, pp. 957–960.

[12] M. K. You and G. Y. Song, ‘‘Case study: Co-simulation and co-emulation environments based on SystemC & SystemVerilog,’’ in *Proc. IEEE Region 10 Conf.*, Jan. 2009, pp. 1–4.

[13] Accellera Systems Initiative, ‘‘Standard universal verication methodology class reference manual, release 1.2,’’ Accellera Syst. Initiative, Napa, CA, USA, Tech. Rep. UVM 1.2, 2014. Accessed: Feb. 17, 2020. [Online]. Available: https://accellera.org/images/downloads/standards/uvm/UVM_Class_Reference_Manual_1.2.pdf

[14] J. Podivinsky, M. Imkova, O. Cekan, and Z. Kotasek, ‘‘FPGA prototyping and accelerated verification of ASIPs,’’ in *Proc. IEEE 18th Int. Symp. Design Diag. Electron. Circuits Syst.*, Apr. 2015, pp. 145–148.

[15] Y.-N. Yun, J.-B. Kim, N.-D. Kim, and B. Min, ‘‘Beyond UVM for practical SoC verification,’’ in *Proc. Int. SoC Design Conf.*, Nov. 2011, pp. 158–162.

[16] H. Zhaohui, A. Pierres, H. Shiqing, C. Fang, P. Royannez, E. P. See, and Y. L. Hoon, ‘‘Practical and efficient SOC verification flow by reusing IP testcase and testbench,’’ in *Proc. Int. SoC Design Conf. (ISOCC)*, Nov. 2012, pp. 175–178.

[17] R. Edelman and R. Ardeishar, ‘‘UVM SchmooVM-I want my C tests,’’ in *Proc. Design Verification Conf. Exhib. (DVCON)*, Mar. 2014, pp. 1–10.

[18] J. Curreri, G. Stitt, and A. D. George, ‘‘High-level synthesis techniques for in-circuit assertion-based verification,’’ in *Proc. IEEE Int. Symp. Parallel Distrib. Process., Workshops Phd Forum (IPDPSW)*, Apr. 2010, pp. 1–8.

[19] L. Gong and O. Diessel, *Verification Challenges*. Cham, Switzerland: Springer, 2015, ch. 2, pp. 15–40, doi: 10.1007/978-3-319-06838-1_2.

[20] M. Karlesky, M. VanderVoord, and G. Williams, ‘‘A simple unit test framework for embedded C,’’ Unity Project, Tech. Rep., 2012. Accessed: Feb, 18, 2020. [Online]. Available: https://dh8.kr/workshop/datastructure/linked_list/vendor/ceedling/docs/Unity%20Summary.pdf

[21] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ, USA: Prentice-Hall, 2003.

[22] N. Dalal and B. Triggs, ‘‘Histograms of oriented gradients for human detection,’’ in *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. (CVPR)*, vol. 1, Jun. 2005, pp. 886–893.

[23] J. Barba, F. Rincon, F. Moya, F. J. Villanueva, D. Villa, J. Dondo, and J. C. Lopez, ‘‘OOCE: Object-oriented communication engine for SoC design,’’ in *Proc. 10th Euromicro Conf. Digit. Syst. Design Architectures, Methods Tools (DSD)*, Aug. 2007, pp. 296–302.

[24] G. Coulouris, J. Dollimore, and T. Kindberg, *Distributed Systems: Concepts and Design (International Computer Science)*, 4th ed. Longman, The Amsterdam: Addison-Wesley, 2005.

[25] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii, ‘‘CHStone: A benchmark program suite for practical C-based high-level synthesis,’’ in *Proc. IEEE Int. Symp. Circuits Syst.*, May 2008, pp. 1192–1195.

JULIÁN CABA received the M.S. degree in computer science and the Ph.D. degree from the University of Castilla-La Mancha (UCLM), Spain, in 2009 and 2018, respectively. He is currently an Assistant Professor with the TSI Department, UCLM. His current research interests include hardware verification methodologies, high-level synthesis, run-time reconfigurable systems, and heterogeneous distributed systems. He won the Ph.D. category at the Xilinx Open Hardware Contest, in 2017.

FERNANDO RINCÓN (Member, IEEE) received the degree in computer science from the Autonomous University of Barcelona, Barcelona, Spain, in 1993, and the Ph.D. degree from the University of Castilla-La Mancha, Ciudad Real, Spain. He is currently an Assistant Professor with the TSI Department, University of Castilla-La Mancha. His research interests include system-on-chip integration, HW run-time reconfiguration, and heterogeneous distributed systems.

JESÚS BARBA received the M.S. and Ph.D. degrees in computer engineering diploma from the University of Castilla-La Mancha (UCLM), Spain, in 2001 and 2008, respectively. He has been working as an Associate Professor with the Department of Information and Systems Technology (TSI), since 2001. He is a member of the ARCO Research Group with the School of Computer Science, UCLM. Among the open research lines and interests it is worth mentioning the following, such as "Low-cost andlow power reconfigurable systems for ubiquitous computing," "Reconfigurable computing platforms for AI algorithms," "Heterogeneous distributed embedded computing," and "High-level synthesis tools."

JOSE A. DE LA TORRE received the M.S. degree in computer science from the University of Castilla-La Mancha (UCLM), Spain, in 2016. He is currently pursuing the Ph.D. degree under an FPU scholarship awarded from the Ministry of Education. His current research interests include deep neural networks, heterogeneous distributed systems, and embedded systems. He received the Extraordinary Degree Award from the UCLM.

JULIO DONDO graduated in electrical-electronic engineering and received the M.Sc. degree in software engineering from the National University of San Luis, San Luis, Argentina, in 1996 and 2007, respectively, and the Ph.D. degree from the University of Castilla-La Mancha, Ciudad Real, Spain, in 2010. He is currently working as an Associate Professor with the National University of San Luis. His research interests include integration of complex embedded systems, reconfigurable computing, heterogeneous distributed systems, and reconfigurable grid computing.

JUAN C. LÓPEZ (Member, IEEE) received the M.S. and Ph.D. degrees in telecommunication (electrical) engineering from the Technical University of Madrid, Madrid, Spain, in 1985 and 1989, respectively.

From September 1990 to August 1992, he was a Visiting Scientist with the Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, USA. From 1989 to 1999, he was an Associate Professor with the Department of Electrical Engineering, Technical University of Madrid, Madrid, Spain. In 1999, he joined as a Full Professor of Computer Architecture with the University of Castilla-La Mancha, Ciudad Real, Spain. His research interests include embedded system design, distributed computing, and advanced communication services.