Invited Paper: UVVM — The Fastest Growing FPGA Verification Methodology Worldwide!

Espen Tallaksen
Bitvis AS, *A Company in the Acando Group*
Norway
espen.tallaksen@bitvis.no

***Abstract*—On average half the development time for an FPGA is spent on verification. It is possible to significantly reduce this time, and major reductions can be accomplished with only minor adjustments and no extra cost. For an FPGA design we all know that the architecture — all the way from the top to the micro architecture — is critical for both the FPGA quality and the development time. It should really be obvious that this also applies to the testbench. UVVM (the open source Universal VHDL Verification Methodology) was developed to solve this and will reduce the verification time significantly while at the same time improving the product quality. UVVM provides a very simple and powerful architecture that allow designers to build their own test harness and test cases much faster than ever before. This paper will show you how simple it is to understand and get started. It will also show how UVVM will help you making better VHDL testbenches and at the same time reduce the workload.**

I. Introduction

Verification constitutes 50% of the total workload for an FPGA project, and the percentage is increasing, but equally interesting is the fact that on average nearly half of the verification time is spent on debugging [1]. This means that there is a huge improvement potential on verification in general and debugging in particular. Hence debugging support is critical for verification efficiency. Another very important factor for schedule and workload is of course efficient reuse. This paper will explain why UVVM [2] is the best possible approach to VHDL [3] verification and debugging and testbench reuse, but first a brief and important digression.

II. Efficient testbench development with VHDL

All the major technology and tool vendors are talking about the “verification gap” and provide “solutions” on how to solve this. Unfortunately, their preferred solution is often a very complex methodology called UVM (Universal Verification Methodology) [4], which involves also learning a new language (SystemVerilog [5]) and buying very expensive tools. An overwhelming majority of FPGA designers in Europe are using VHDL as their design and verification language (60% and 50% world-wide respectively [1]) and there is no reason to change that.

UVVM has actually copied a few of the principles from UVM and could thus be seen as a simplified VHDL version of UVM, only in a familiar language, being component oriented and simpler to use and understand. VHDL is powerful enough for almost all FPGA projects — and allows those developers to continue using a language they already know, and then step by step just add new functionality as needed. In other words, a gentle and efficient evolution on your current testbench and language, but features and functionality alone however are not sufficient: structure and architecture are key to absolutely all important aspects of good FPGA development.

People often talk about efficient code writing, but this is the wrong focus. Code writing is only a very small part of the complete module or FPGA development — with the difference from one approach to another having a relatively small effect on the overall development efficiency. What really matters is:

- Overview: Easy understanding of the test harness and how the testbench works.
- Modifiability, Maintainability and Extensibility: Ease of modifying or extending the functionality.
- Debuggability: Good support for finding the root cause of a problem
- Reusability: Efficient reuse between module TBs (Test-Benches), from module TBs to FPGA top level TB, from one project to another, and between verification modules (aka Verification Components).

If we now go one step down into the details, we see that some functionality is **always** required for **any** good testbench:

- Alert handling: To assure that any potential problem is flagged, and a proper alert message is given. Should also allow alerts to be counted, summarised and manipulated in various ways.
- Logging: To allow simple messages from specific activities, but also detailed messages on progress. Could reduce debug time to a fraction, when prior to an alert a detailed transaction overview is shown in a structured manner.
- Checking signal values: The most common thing to do in any TB, e.g. checking outputs.
- Checking signal time aspects: Important for instance to assure that no spikes have occurred, or that a signal has been stable for a minimum period of time.
- Waiting for values or changes: Needed for instance to wait for an interrupt or a given word value. Should have a timeout to prevent waiting forever.

All of this and much more is supported by the UVVM Utility Library, which is the entry level to UVVM and also the basis for the more advanced features. In this paper, we will illustrate how easy this is to use.

This work was presented at the first Workshop on Open-Source Design Automation (OSDA), 29 March 2019, in Florence, Italy. https://osda.gitlab.io

```
-- In test sequencer as a normal progress msg
log("Checking Registers in UART");
```

```
BV: 160 ns    uart_tb    Checking Registers in UART
```

```
-- In test sequencer as a section header
log(ID_LOG_HDR, "Check defaults for all registers");
```

```
BV:   60 ns    uart_tb    Check defaults for all registers
BV:-----------------------------------------------------
```

Fig. 1: Simple log commands and resulting transcript

```
-- E.g. inside the test sequencer
check_value(dout, x"00", ERROR, "dout must be default inactive");
```

```
BV: 60 ns   irqc_tb   check_value(slv x00)=> OK.
                      dout must be default inactive
```

```
BV:=========================================================
BV: ERROR:
BV:      192 ns. irqc_tb
BV:                value was: 'xFF'.  expected 'x00'.
BV:                dout must be default inactive
BV:=========================================================
```

Fig. 2: Checking a value and resulting transcripts for positive acknowledge and mismatch

III. Example UVVM Usage

Some command examples are illustrated in the Figures 1 and 2. Logging and value checking has been chosen as examples as these are two of the most important functions. Fig. 1 shows the VHDL command used (in grey) followed by the resulting transcript (in blue) when executing these two variants of the `log()` command in a simulator (The prefix '`BV`' may be modified by the user).

Several more advanced versions of the log procedure are also available, allowing for instance verbosity control and more information, but this can be found from the provided quick reference when needed.

Numerous variants of the `check_value()` command are provided, for different signal types, with and without a return value, and several more advanced options not needed for beginners. Figure 2 shows the command and the resulting transcript — in blue for a positive acknowledge transcript — and in red for a failing check. The mismatch report here is really useful for debugging.

There are also lots of other very useful commands in the UVVM Utility Library. These can all be found in the easy to use Quick Reference provided with the library. In this PDF document you will first find an overview of all command types and commands, and then every single command is properly documented with overloads, examples, description and comments. The `await_value()` command is shown in Figure 3 as only one example of the many commands on offer, whereas the description in the Quick Reference for this command is shown in Figure 4.

```
await_value(irq, '1', 0 ns, 2* C_CLK_PERIOD,
    ERROR, "Interrupt expected immediately");
```

Fig. 3: Waiting for an event to happen

In this example the process (for instance the test sequencer) calling this command will wait for the signal '`irq`' to go to '`1`', at the earliest after 0 ns and at the latest after two clock periods. Otherwise it will fail, or time out and fail, respectively. In both cases it will fail with an error message that includes the provided string describing the intention for this specific command.

It is important that a testbench always reports at least pass or fail, but preferably also a general summary of all the alerts, as shown in Figure 5.

A. *Getting started with UVVM Utility Library*

As previously mentioned, all the above examples and lots of other very useful commands are part of the entry level of UVVM — the testbench infrastructure called the UVVM Utility Library. You can download UVVM from github.com/UVVM/UVVM, and there you can also find more detailed introductions to the Utility Library and the various other parts of UVVM.

Feedback from all UVVM users so far are unanimous in saying that using the Utility Library is as easy as it gets.Below is the **exhaustive** list of what to do to get started — and it takes less than 5 minutes to go through:

1) Download from GitHub:
   https://github.com/UVVM/UVVM
2) Compile the Utility Library:
   a) Inside your simulator go to *'uvvm_util/script'*
   b) execute: *'do ../script/compile_src.do'*
3) Include the library inside your testbench by adding the following lines before your testbench entity declaration:
   `library uvvm_util;`
   `context uvvm_util.uvvm_util_context;`
4) You may now enter any utility library command inside your testbench processes (or subprograms); e.g. `log("Hello world");`

Note that UVVM has been tested with Modelsim, Riviera-PRO and GHDL, and should thus also work with Questa and Active-HDL, and any other VHDL-2008 compatible simulator. The script mentioned above is tested with Modelsim. GHDL provides dedicated UVVM scripts.

IV. Design Structure and Architecture

Figure 6 shows again the most critical factors for both design and verification throughout a project. During the design phase engineers are very much aware of this, which is the main reason designs are split into more or less autonomous modules with stand-alone functionality, like for instance a UART, SPI, DMA controller, ADC, etc.

These modules (or VHDL entities/components) are given commands from software, and then they perform their requested tasks while the software has continued executing and is doing something completely different in parallel. If anything goes wrong or the module needs attention, it just screams out — normally via an interrupt. The nice thing about this approach is that the FPGA top level design is then very easy to understand.

```
target(bool), exp(bool), min_time, max_time, alert_level, msg, [scope, (etc.)]]
target(sl),   exp(sl),   [match_strictness], min_time, max_time, alert_level, msg, [scope, (etc.)]
target(slv),  exp(slv),  [match_strictness], min_time, max_time, alert_level, msg, [scope, (etc.)]
target(u),    exp(u),    min_time, max_time, alert_level, msg, [scope, (etc.)]
target(s),    exp(s),    min_time, max_time, alert_level, msg, [scope, (etc.)]
target(int),  exp(int),  min_time, max_time, alert_level, msg, [scope, (etc.)]
target(real), exp(real), min_time, max_time, alert_level, msg, [scope, (etc.)]
```

Examples

```
await_value(bol, true, 10 ns, 20 ns, ERROR, "Waiting for bol to become true");
await_value(slv8, "10101010", MATCH_STD, 3 ns, 7 ns, WARNING, "Waiting for slv8 value");
```

Waits until the *target* signal equals the *exp* signal, or times out after *max_time*.

An alert is asserted if the signal does not equal the expected value between *min_time* and *max_time*.

Note that if the value changes to the expected value at exactly *max_time*, the timeout gets precedence.

- *match_strictness:* Specifies if match needs to be exact or std_match, e.g. 'H' = '1'. (MATCH_EXACT, MATCH_STD)

Defaults:
***match_strictness*<=MATCH_EXACT,**
***scope*<=C_TB_SCOPE_DEFAULT,**
***msg_id*<=ID_POS_ACK,**
***msg_id_panel*<=shared_msg_id_panel**

Fig. 4: Utility library quick reference for the await_value command. Procedure parameters are shown in the left column with one line per procedure overload, followed by two examples. Explanations are given in the right column

```
==============================================================
BV:  ***  SUMMARY OF ALL ALERTS  ***
BV:  ==========================================================
BV:                    REGARDED  EXPECTED  IGNORED   Comment?
BV:    NOTE         :     0         0         0      ok
BV:    TB_NOTE      :     0         0         0      ok
BV:    WARNING      :     0         0         0      ok
BV:    TB_WARNING   :     0         0         0      ok
BV:    MANUAL_CHECK:      0         0         0      ok
BV:    ERROR        :     0         0         0      ok
BV:    TB_ERROR     :     0         0         0      ok
BV:    FAILURE      :     0         0         0      ok
BV:    TB_FAILURE   :     0         0         0      ok
BV:  ==========================================================
BV:  No mismatch between counted and expected serious alerts
BV:  ==========================================================
```

Fig. 5: Transcript of alert summary

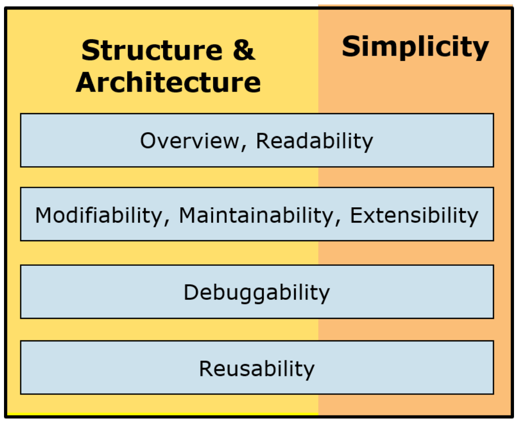

Fig. 6: Critical factors for Design and Verification

We see the modules and know their functionality. The bus system to the modules is standardised (e.g. AXI4-lite [6] or Avalon [7]) — the bus interface of the modules is standardised to fit the bus — and the software is running high level commands to control it all.

V. Verification Structure and Architecture

It is obvious that the design structure explained above is both extremely efficient and promotes good quality. For testbenches however, there has been no similar system or methodology. This is one major reason most testbenches are rather unstructured, and the main reason UVVM was developed.

In UVVM the design architecture is mirrored, resulting in a testbench architecture that is really easy to understand, and keeping all the benefits of a good modular structure:

- Verification modules with a well understood functionality and interface
- A standardised bus system from the sequencer to the verification modules
- A standardised bus interface on all the verification modules

In UVVM these verification modules are called VVCs (VHDL Verification Components). In Figure 7 'SBI_VVC' is just a Simple Bus Interface, but could equally well be 'AXI4-lite' or 'Avalon'.

The UVVM architecture is very similar to the FPGA design architecture with an external software sequencer mentioned above. The test (case) sequencer distributes commands to the VVCs the same way software distributes commands to FPGA modules. Then the VVCs do what they are told — in the same way as the modules in an FPGA. This normally means handling an access on their physical interface. For the FPGA module that could be to transmit some data, and in fact that could also be the case for the VVC. For the UART testbench in Figure 7, the test sequencer could typically send a command to the UART_TX_VVC to transmit a byte onto the RX input of the UART.

VI. Test Sequencer Interface

In UVVM there are two ways to access an interface of the DUT (Device Under Test). The simplest form is just a basic BFM (Bus Functional Model) procedure as shown in Figure 8, where the last parameter serves as a code line comment, and is written to the transcript when needed. This procedure will be executed directly from the process from which it was called, and apply data `x4C` onto UART RX — using the UART protocol. This means if the test sequencer is calling this procedure, it will apply the UART protocol bit by bit. This will take some time — during which the test sequencer will be busy and cannot do anything else.

The more advanced form is what is called a CDM (Command Distribution Method) as shown in Figure 9. In this case when the sequencer calls this procedure — the command is instantly sent to the VVC given by the target address in red — giving the name and the instance number of the VVC that should actually execute the command onto the DUT. The transfer of

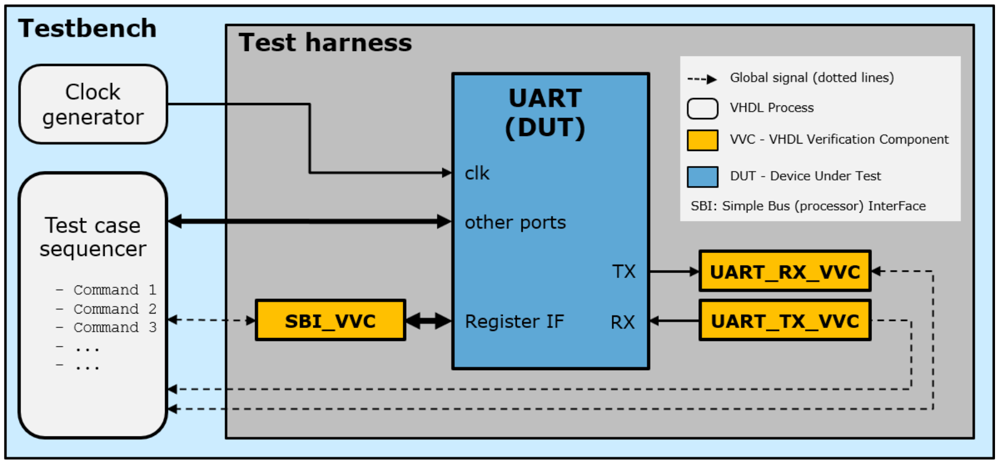

Fig. 7: Structured testbench architecture

```
uart_transmit(x"4C", "Apply byte on UART RX");
```

Fig. 8: UART transmission command — as a BFM

```
uart_transmit(UART_TX_VVCT,1  x"4C", "Apply byte on UART RX");
```

Fig. 9: UART transmission command — using VVCs

the command from the test sequencer to the VVC takes no time, which means the test sequencer is immediately free to do anything else — like initiating an access on another interface. The time consuming UART protocol will be taken care of by the VVC.

A. *A complete small test*

The test sequencer code lines in Figure 10 show how a very small test can be written to check simultaneous access on all interfaces. This example is from a slightly different testbench than above, as now the VVCs for UART TX and RX are combined into a single VVC for better reuse. For this case we decided to add another target parameter to indicate the channel (RX vs TX).

B. *Microarchitecture is critical*

In any FPGA design it is great, but not sufficient to have a good top-level architecture. It is equally important to have a good microarchitecture all the way down. This applies of course also to the testbench, and in this case it means that the architecture of the VVCs must also be easy to understand, modify and reuse. And again — standardisation would be great.

```
sbi_write(SBI_VVCT,1, C_ADDR_TX_DATA, x"A0", "Send byte UART TX");
uart_expect(UART_VVCT,1,RX  x"A0", "Check byte from UART TX");
uart_transmit(UART_VVCT,1,TX  x"A1", "Apply byte on UART RX");
wait for C_FRAME_PERIOD;
sbi_check(SBI_VVCT,1, C_ADDR_RX_DATA, x"A1", "Check UART RX byte");
```

Fig. 10: A complete small test — using VVCs

The basic VVC consists of the blocks in yellow in Figure 11, and this applies to absolutely **all** VVCs. The VVC will see the CDM above and only accept commands that are targeted for exactly that VVC. Then it will put the `uart_transmit()` command on the queue and allow the test sequencer to continue. The Executor will fetch the command from the Queue, see that this is a `uart_transmit()` command, and then execute the BFM towards the DUT using the provided parameters.

VII. Structured VVC extensions

The very structured architecture of the VVCs has an additional benefit. It is very easy to add more structured functionality — either for more advanced interfaces — or just to include more reusable functionality — like various checkers. A split transaction protocol like Avalon MM often results in chaotic testbenches, as there could for instance be multiple read requests before getting the first read response. With the VVC architecture shown in Figure 11 this is quite simple and very structured. The base executor fetches the read command from the queue and knows that this is a split transaction access. It thus calls the read request BFM procedure, and at the same time passes the read command on to the next queue. The response executor will fetch this and then just call the read response BFM procedure and wait for the response. This way there could be any number of read requests between any number of read responses, and all is still nice and structured.

In UVVM a script is provided to generate a VVC for new interfaces. The generated code is a template, and it is clearly marked where the users need to add their own BFMs and various adaptations. Given the BFM procedures, which the users need to make for any testbench anyway, it only takes about 30 minutes to make a complete VVC from scratch. According to Doulos [8] — slightly longer the first time, but probably 30 minutes the next time.

In summary, the most important parts of a testbench are the following:

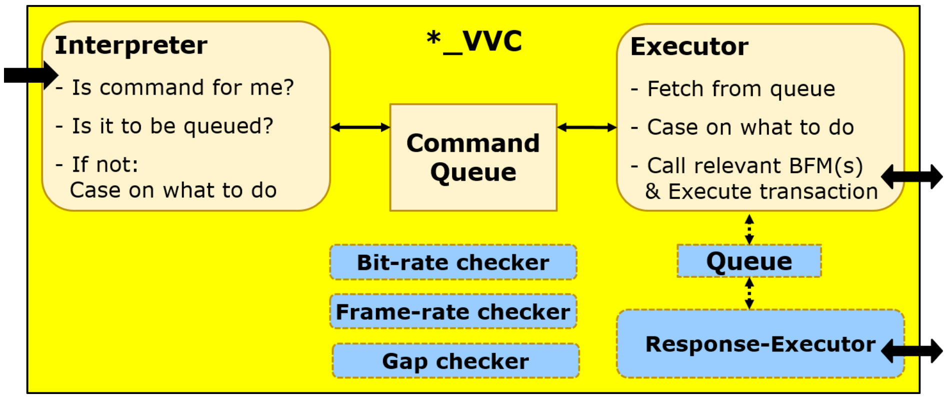

Fig. 11: Structured VVC architecture

- The basic testbench infrastructure — with log, alert, checks, awaits, etc. and potentially BFMs to handle interface accesses.
- The top-level test harness — for getting the total overview.
- The Verification modules (when needed) — understanding and generating these.
- The test cases — writing the actual tests and understanding the complete sequence of events.

The first bullet is handled by the Utility library. The next two bullets are taken care of through a very structured architecture — both at the top level and inside the VVCs, whereas the last and probably most important bullet — as this is where you will spend most of the time — is taken care of by allowing high level, easily understandable commands control the complete testbench — including simultaneous activity on multiple interfaces.

VIII. Standardisation

For FPGA design, standardising over several decades on a good architecture concept with control and status registers, a bus system, autonomous modules and high-level software commands has resulted in an efficient FPGA design methodology, but also efficient software development — as the software developers do not need to understand the details of the modules they are controlling. This may seem obvious — and it is... So why hasn't the same happened for FPGA testbenches, where we have exactly the same scenario, with a sequencer controlling multiple interfaces? This is what UVVM now provides, and has even standardised the most important aspects:

- Standard TB infrastructure
- Standard BFM setup and usage
- Standard module (VVC) control & status interface
- Standard protocol from sequencer to VVCs
- Standard commands for the sequencer (for common tasks)
- Standard VVC internal architecture
- Standard command queuing system
- Standard handling of multithreaded interfaces (e.g. split transactions)
- Standard synchronisation methods between the VVCs (or any other process)
- Standard waiting and timeouts
- Standard messaging (transcript) and alert handling
- Standard multicast and broadcast commands
- Standard debugging support
- Standardised testbench reuse

IX. Explosive world-wide growth

International players with a strong interest for VHDL and verification are now embracing UVVM. The two main VHDL simulator providers — Aldec and Mentor Graphics — are both showing great interest, with Aldec already running several webinars [9], [10] on UVVM and Mentor including UVVM presentations in their 'FPGA verification day' [11]. Doulos, the number one international training provider for FPGA and ASIC methodology, are now recommending UVVM for VHDL testbench architecture [8]. A final piece of evidence that UVVM is really gaining momentum is that ESA (European Space Agency) is now backing UVVM through a contract of more than 250k euro for an extension of UVVM, and may recommend their suppliers to use UVVM for FPGA verification (unless they already have a good internal solution) [12].

Gaining momentum is actually a major understatement. From 2016-Q2 to 2018-Q2 (between two Wilson Research Group studies) the number of UVVM users world-wide has exploded — from a fraction of a percent in 2016 to 10% of all FPGA developers in 2018-Q2 [1]. This is beyond our wildest expectations, and it keeps on growing...

UVVM runs perfectly well with any VHDL-2018 compatible simulator, like the ones from Aldec and Mentor Graphics. For the open source community it will probably be good to hear that GHDL [13] has now (from February 2019) been updated to work with UVVM. GHDL also provides scripts to compile UVVM for GHDL.

X. On-going and future extensions

The ESA-project will provide several new and important features to UVVM. Some have already been released, like an easy-to-use, flexible and generic Scoreboard, and some are still being developed, like watchdogs, error insertion, better controlled randomisation and functional coverage, and more support for connecting DUT models to the testbench harness.

The most important feature to come soon is hierarchical verification components. This will take care of the challenge of using scoreboards to verify data going through a DUT when the communication layer is not the same on both sides, for instance when the input on one side is byte or word oriented, like AXI4-lite or Avalon, and the output on the other side is packet oriented, like Ethernet. Hierarchical verification components will allow such scenarios to be handled in a simple way.

UVVM is also being continuously updated with minor functionality outside the ESA project, and plans are currently being made for further both small and large extensions.

XI. VHDL community and UVVM licensing

UVVM is open source using the MIT license. This license is among the most open and permissive licenses used for open source. This gives UVVM users a major freedom in using UVVM and making their own BFMs, VVCs or other UVVM compatible verification IP.

In fact a new repository [14] has just been opened for external contributions. The architecture and standardisations mentioned above, allow UVVM compatible VVCs from any company and designer to be used together in the same testbench — working the same way and working together.

Commercial use of UVVM is permitted — with no restrictions other than those imposed by the permissive MIT license. Commercial VVCs could indeed be a great contribution to the VHDL and UVVM community as that would increase the number of available verification IP. Quite a few UVVM users have privately stated that they would be very much willing to pay for such IP.

XII. Conclusion

Structured verification is very important for efficiency, quality and reuse. UVVM is free and open source and provides a very structured methodology and library. UVVM even comes with free and open source VVCs and BFMs for UART, I2C, SPI, SBI, AXI4-lite, AXI4-stream, GPIO and Avalon MM. This allows a great kick-start for any project using these (or similar) peripherals.

We are now ready for the next chapter, with further development of UVVM, new add-on features, and our new mechanism for sharing VVCs within the VHDL community — both open source and commercial.

With the explosive growth in number of users, the new features, the increasing interest from the big vendors, and the fast-growing attention among VHDL users — together with the game changing simplicity, overview, readability, maintainability, extensibility and reuse-friendliness — UVVM will keep on growing with a huge momentum in the FPGA community.

References

[1] H. Foster, “The 2018 Wilson Research Group ASIC and FPGA Functional Verification Study,” https://www.mentor.com/products/fv/events/the-2018-wilson-research-group-asic-and-fpga-functional-verification-study.

[2] Bitvis AS, “UVVM,” https://github.com/UVVM/UVVM.

[3] “IEEE Standard VHDL Language Reference Manual,” *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, Jan 2009.

[4] “IEEE Standard for Universal Verification Methodology Language Reference Manual,” *IEEE Std 1800.2-2017*, May 2017.

[5] “IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language,” *IEEE Std 1800-2017 (Revision of IEEE Std 1800-2012)*, Feb 2018.

[6] Xilinx, “AXI Reference Guide,” https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf, Mar 2011.

[7] Intel, “Avalon® Interface Specifications,” https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl_avalon_spec.pdf, Sep 2018.

[8] Doulos, “OSVVM and UVVM - VHDL Verification Methodology,” https://www.doulos.com/content/events/OSVVM_UVVM_VHDL.php.

[9] Aldec, “UVVM - A game changer for FPGA VHDL Verification,” https://www.aldec.com/en/downloads/private/920.

[10] ——, “Universal VHDL Verification Methodology (UVVM) The standardized open source VHDL testbench architecture,” https://www.aldec.com/en/downloads/private/1158.

[11] “FPGA Verification Day 2019,” https://trias-mikro.de/en/dates/fpga-verification-day.

[12] E. Tallaksen, “UVVM — Universal VHDL Verification Methodology. Setting a standard for VHDL testbenches.” SEFUW: SpacE FPGA Users Workshop, 4th Edition; https://indico.esa.int/event/232/contributions/2159.

[13] GHDL, https://github.com/ghdl/ghdl.

[14] UVVM Community VIPs, https://github.com/UVVM/UVVM_Community_VIPs.