ChiselVerify: An Open-Source Hardware Verification Library for Chisel and Scala

Andrew Dobis*, Tjark Petersen*, Hans Jakob Damsgaard*, Kasper Juul Hesse Rasmussen*,
Enrico Tolotto*, Simon Thye Andersen*, Richard Lin†, Martin Schoeberl*

*Department of Applied Mathematics and Computer Science
Technical University of Denmark
Lyngby, Denmark

†Department of Electrical Engineering and Computer Sciences
UC Berkeley
Berkeley, CA

adobis@student.ethz.ch, s186083@student.dtu.dk, hans.damsgaard@tuni.fi, s183735@student.dtu.dk,
s190057@student.dtu.dk, simon.thye@gmail.com, richard.lin@berkeley.edu, masca@dtu.dk

***Abstract*—Modern digital hardware is becoming ever more complex. The development of different application-specific accelerators rather than traditional general purpose processors calls for advanced development methods not only for design, but equally so for subsequent verification. Recently, this has made engineers propose an agile hardware development flow. However, one of the main obstacles when proposing such a method is the lack of efficient tools.**

Chisel, a high-level hardware construction language, was introduced in order to combat this lack. Since this already enables agile hardware design, we instead focus our attention on the verification flow. Thus, this paper proposes ChiselVerify, an open-source library for verifying circuits described in Chisel. It builds on top of Chisel and uses Scala to drive the verification process. The solution is well integrated into the existing Chisel universe, making it an extension of currently existing testing libraries.

***Index Terms*—digital design, verification, Chisel, Scala**

I. Introduction

Over the past several years, hardware design has grown to be ever more complex. The increased demand for high-performance computing systems has lead to a larger need for domain-specific hardware accelerators [1]. The design of these accelerators is often complex, and their development is time-consuming and error-prone. In order to combat this added time-constraint, we can learn from software development trends such as agile software development [2], and adapt to agile hardware development [3]. Chisel [4], a Scala-embedded hardware construction language, was introduced in order to move digital circuit description to a more software-like high-level language.

Hardware design is dominated by the traditional hardware description languages (HDLs), Verilog and VHDL, and the more recent SystemVerilog. But while SystemVerilog does extend Verilog with object-oriented features for verification, its hardware description flow remains the same as with Verilog. Thus, it does not fit an agile development flow. Chisel attempts to solve these issues by providing full support for functional and object-oriented programming. However, Chisel is missing efficient verification tools with limited functionality available in the corresponding ChiselTest package [5].

As such, we choose to base our work on Chisel and ChiselTest, and aim to raise the tooling level for a digital design. We have developed a verification framework inspired by the Universal Verification Method (UVM), but implemented by leveraging Scala's conciseness and support for both object-oriented and functional programming. Our framework, ChiselVerify, supports both coverage-oriented and constrained random verification (CRV) flows with more features than those available in UVM.

As a showcase, we have verified an industrial use case, a min-heap, utilizing ChiselVerify to check as many features of the min-heap with as few lines of verification code as possible.

The main contribution of this paper is ChiselVerify [1], an open-source verification library for hardware designs.

The paper is organized into 6 sections. Section II describes related work. Section III describes background on hardware verification. Section IV describes our solution for enabling verification in Chisel, namely ChiselVerify. Section V explores ChiselVerify on an industry-provided use case. Section VI concludes.

II. Related Work

SystemVerilog is an extension of Verilog. Many non-synthesizable extensions are intended to write more advanced

978-1-6654-0712-0/21/$31.00 ©2021 IEEE

[1] https://github.com/chiselverify/chiselverify

test-benches. SystemVerilog adds object-oriented programming for those test-benches. However, in contrast to Chisel, the object-oriented addition cannot be used for hardware description. SystemVerilog offers certain constructs capable of gathering coverage information [6], such as statement and functional coverage. When it comes to functional coverage, our solution differs in several ways from SystemVerilog. On top of range-based bins, ChiselVerify's `cover` constructs can take temporal relations into account, as well as generalized conditional bins that work using purely user-defined hit predicates. This differs from SystemVerilog, which mainly focuses on bins that cover value ranges or transitions.

The *Universal Verification Methodology* (UVM) was created as a standardized way of writing test-benches on top of SystemVerilog. It allows for the creation of reusable test-benches (i.e., using the same test for multiple designs) [7]. However, it is inherently verbose, since even a simple test requires around 800 lines of SystemVerilog code. UVM thus requires a significant initial time-investment, but is reusable once it gets up and running. UVM's structure differs from most traditional test-benches, making it less accessible for newcomers than the simpler approach done by ChiselTest.

Other projects have also focused on applying software testing techniques to hardware verification. RFuzz [8] focuses on creating a generalized method that enables efficient "coverage-guided fuzz mutational testing". This method relies on FPGA-accelerated simulation and new solutions allowing for quick and deterministic memory resetting, to efficiently use fuzzing (i.e., randomized testing, where the random seeds are mutated depending on certain coverage results) on digital circuits. The coverage metrics used in this solution are automated and based on branch coverage. This work offers a different type of solution. While we work mostly on verification functionalities inside a language, RFuzz delivers an efficient way to use said functionalities in order to ameliorate testing. RFuzz uses functional coverage tools in order to guide its randomized testing. Current work is also being done, in the scope of the ChiselVerify project, on coverage driven mutational fuzzing for digital circuits [9], however this is out of the scope of this paper.

`Chisel3.formal` is a formal verification package containing a set of tools and helpers for formally verifying Chisel modules [10]. In contrast to ChiselVerify, `chisel3.formal` proposes way of testing based around defining a set of formal checks that a design must pass in order to be considered as correct. These checks can, for example, look like: `past(io.out, 1) (pastIoOut => { assert(io.out >= pastIoOut) })` which guarantees that the current module will never decrease its output from one cycle to the next. These formal checks can then be verified by calling the `verify(module)` function.

This approach is similar to software contracts in Scala, like the ones enabled by ScalaCheck [11]. The main difference between our solution and this one is that here the rules are written on a per-module basis and are thus directly linked to the Chisel code, while our solution focuses on checking that a suite of test-benches is testing the right things. The `chisel3.formal` package has also been extended in `kiwi-formal` [12] and `dank-formal` [13], each adding their own additional formal rule templates.

As far as we know, ChiselVerify is the only verification framework allowing for the easy use of verification functionalities, well integrated into the ChiselTest-Chisel ecosystem.

III. Background

This section presents a brief overview of what hardware verification is. We also briefly present Chisel and the current solutions that exist in it with regards to the verification of digital designs.

A. Verification of Digital Designs

Verification of digital designs refers to the testing of a design before it has been taped-out [6]. SystemVerilog [14] is one of the main languages used for verification. The language enables verification engineers to define constraint-driven randomized test-benches and define metrics to gather functional coverage data related to a test suite. However, being embedded in a low-level language makes writing tests quite complex. The three main verification features that we are interested in are: functional coverage, constrained random verification, and bus functional modeling.

1) Functional Coverage: One of the main tools used in verification is test coverage. This allows verification engineers to measure their progress throughout the testing process and understand how effective their tests are. In contrast to the more common statement coverage, which defines a quantitative measure of the testing progress *"How many lines of code have been tested?"*, functional coverage gives a qualitative measure, *"Which functionalities have we tested?"* [6]. Functional coverage enables the measurement of how correctly the specification has been implemented. This is measured relative to a verification plan, which includes the following components:

- `Bins`: Ranges of values that should be tested for (i.e., expected values of a given port).
- `Cover` constructs: Ports that need to be sampled in the coverage report, defined using a set of `bins`.

2) Constrained Random Verification: CRV allows the verification engineer to create random variables which generate values that satisfy a set of associated constraints. With constrained random inputs, a relatively small test suite can, statistically, cover many of a component's functionalities. In addition, the constraints help to ensure that no invalid input combinations that would not appear during regular operation are applied [15].

These constraints define a constraint satisfaction problem (CSP). CSP represents the entities of a problem as a finite homogeneous collection of constraints. CSP solvers thus serve as the basis for generating sets of constrained random signal values for verification.

SystemVerilog has native support for constrained random data types and a built-in CSP solver. Variables declared with

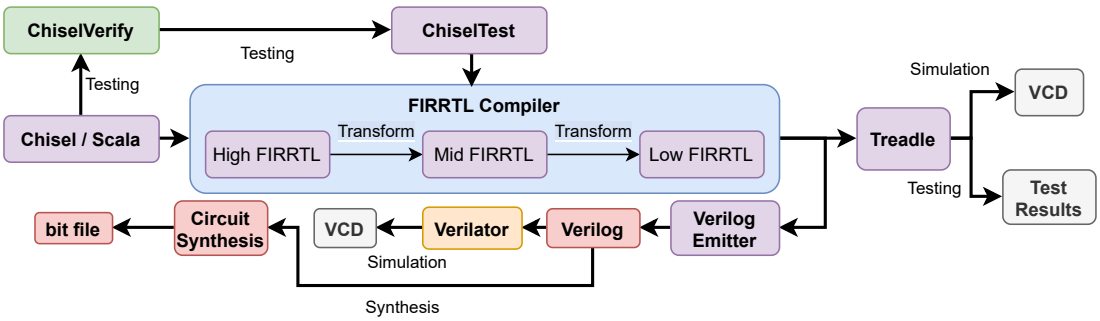

Fig. 1: Overview of the Chisel compilation pipeline.

the `rand` keyword are randomizable upon calling a `randomize` method.

3) Bus Functional Models: A bus functional model (BFM) is an abstract model of a (standardized) interface that enables interacting with manager or subordinate components at a transaction level rather than at the level of individual wires. Many synthesis tools, including Xilinx's Vivado, provide IP generators whose output IP blocks are equipped with such interfaces. BFMs thus enable simpler, safer, and less verbose interactions with interfaces like, e.g., AXI.

B. Digital Design with Chisel

Our verification library is used for designs described in Chisel. Chisel is a "hardware construction language" embedded in Scala, used to describe digital circuits [4]. This language is more high-level than the traditional hardware description languages, such as VHDL or Verilog, and enables object-oriented and functional programming in the context of digital design.

Since Chisel and Scala are executing on the Java virtual machine (JVM), they have an excellent interoperability with Java. We can therefore leverage a large pool of both Java and Scala libraries for hardware design and verification. Furthermore, the packaging system in Scala/Java simplifies the integration of external components.

Working in the JVM also allows for the use of the Java native interface, which enables JVM based languages to call C functions. This enables co-simulations between Scala testers, Chisel designs, and a C-based golden model. This should allow companies to keep their existing C models, but move their simulation workflow into Scala/Chisel testers.

Chisel translates the hardware description into an intermediate representation called FIRRTL [16]. It then performs multiple optimization stages, called transforms, during which high-level concepts, such as a functional map or vectors, are compiled into lower-level concepts that map onto what we usually see in a Verilog or VHDL description. Once that is done, the newly transformed FIRRTL, called Low FIRRTL, can be used either for simulation, using an execution engine such as Treadle, or for synthesis by translating it into Verilog, which is then used to generate the synthesized circuit. Note that the final Verilog description may also be used for simulation purposes using engines such as Verilator [17]. Figure 1 shows an overview of the Chisel compilation pipeline.

C. Testing Chisel Designs

A digital design described in Chisel can be tested with ChiselTest [5], a non-synthesizable testing framework for Chisel. ChiselTest emphasizes on usability and simplicity while providing ways to scale up in complexity. Fundamentally, ChiselTest is a Scala library that provides access into the simulator through operations like `poke` (write value into circuit), `peek` (read value from circuit), and `step` (advance time). As such, tests written in ChiselTest are just Scala programs, imperative code that runs one line after the next.

ChiselTest is missing fundamental verification functionalities that can improve the verification efficiency of Chisel designs. It is currently not possible to do things such as constrained random testing or obtaining functional coverage results while solely relying on the ChiselTest framework. Functionalities such as those are crucial when it comes to efficiently verify one's design.

IV. Verification with Chisel

We propose ChiselVerify, a verification library written in Scala. ChiselVerify uses the device under test (DUT) interfacing features from ChiselTest in order to enable three main verification functionalities in Chisel: functional coverage, constrained random verification, and bus functional modelling. We also show how our framework can be used to create a bus functional model by creating one for the standardized AXI4 interface. The following subsections explain each functionality and present how to use it.

A. Coverage in Chisel

Our solution enables one to define functional coverage constructs for Chisel designs in Scala. In order to implement the different components needed for functional coverage in Scala, we needed to be able to do the following:

- Define a verification plan, using `cover` constructs.
- Sample DUT ports, using the ChiselTest framework.
- Keep track of bins to sampled value matches, using a coverage database.
- Compile all of the results into a comprehensible coverage report.

Implementing these elements was done using a structure based around a top-level element known as the `CoverageReporter`, enabling one to define a verification plan using a `register` method. This method stores `cover` construct to `bin` mappings inside a `CoverageDB` (DB being short for a database) object. Once the verification plan is defined, ports are sampled using the `sample` method, which is implemented using ChiselTest's peeking capabilities. Finally, at the end of a test suite, a functional coverage report is generated using the `report` method, which compiles the results stored in the database into a Scala `case class` which can be used to obtain coverage percentages and bin hit counts.

```
1 val cr = new CoverageReporter
2 cr.register(
3   cover("accu", dut.io.accu)(
4     bin("lo10", 0 to 9),
5     bin("First100", 0 to 99)),
6   cover("test", dut.io.test)(
7     bin("testLo10", 0 to 9)),
8   cover("accuAndTest", dut.io.accu, dut.io.test)(
9     cross("both1", 1 to 1, 1 to 1))
```

Listing 1: Small Verification Plan defined using 3 `cover` constructs, including one cross coverage construct

Listing 1 is an example of how to define a verification plan using our functional coverage tool. One concept used here is *cross coverage* defined using a `cover` construct on multiple ports. Cross coverage allows one to specify coverage relations between different ports. This means that a cross defined between, e.g., `dut.io.a` and `dut.io.b` will be used to gather information about when `a` and `b` cover specific values simultaneously [6]. In listing 1, we are checking that `accu` and `test` take the value 1 at the same time.

Once our verification plan is defined, we need to decide when we want to sample our cover points using our coverage reporter. This can be done, in our example, simply by calling `cr.sample()` when we are ready to sample our points. Finally once our tests are done, we can ask for a printed coverage report by calling `cr.printReport()` which results in the following:

```
=============== COVERAGE REPORT ===============
================= GROUP ID: 1 =================
COVER_POINT PORT NAME: accu
BIN lo10 COVERING 0 to 9 HAS 8 HIT(S) = 80%
BIN First100 COVERING 0 to 99 HAS 9 HIT(S) = 9%
================================================
COVER_POINT PORT NAME: test
BIN testLo10 COVERING 0 to 9 HAS 8 HIT(S) = 80%
================================================
CROSS_POINT accuAndTest FOR POINTS accu AND test
BIN both1 COVERING 1 to 1 CROSS 1 to 1 HAS
1 HIT(S) = 100%
================================================
```

In the above report, we can see that our two `cover` constructs are listed and that each one of their `bins` has an associated number of hits. This represents how many times the port had a unique value sampled within the given range. A coverage percentage is then given for each bin, representing the ratio between the number of hits and the total number of possible values in the range.

Another element that our framework offers is gathering delayed coverage relationships between two coverage points. The idea is similar to how a `cross` works, but this time rather than sampling both points in the same cycle, we compare one port, at the starting cycle, to another port sampled a given number of cycles later. This number of cycles is called the `delay`, and there are currently three different ways to specify it:

- `Exactly` delay means that a hit will only be considered if the second point is sampled in its range a given number of cycles after the first point was.
- `Eventually` delay means that a hit will be considered if the second point is sampled in its range at any point within the following given number of cycles after the first point was.
- `Always` delay means that a hit will be considered if the second point is sampled in its range during every cycle for a given number of cycles after the first point was sampled.

Finally, we exploit the functional nature of Scala in order to allow for conditional cover points, which offer the possibility to create fully custom hit-consideration rules using a user-defined predicate. This allows the user to check for arbitrary relations between an arbitrary number of ports. One could then, e.g., create a bin that considers a hit every time all fields in a vector are equal. This is defined simply by adding a function of type `Seq[BigInt] => Boolean` to a `cover` construct's `bin`. The report then shows the number of hits that each condition had throughout the test suite. Adding an "expected number of hits" to each condition allows for a final percentage to be shown alongside the number of hits.

These features allow for the definition of complex verification plans that can be used to represent any given specification, making it possible to verify the correct testing of any design. In addition, supporting a coverage measure directly in the testing tool also enables modern verification strategies such as constrained random verification.

B. Constrained Random Verification

To make best use of a coverage-driven verification flow, one needs access to CRV tools. Such tools are, as explained before, included in SystemVerilog, and we provide another implementation in ChiselVerify. ChiselVerify provides a wrapper to an existing CSP solver, named JaCoP [18], and a domain-specific language, which allows users to declare and randomize objects.

1) Constraint Programming with ChiselVerify: To begin writing constrained random objects using our library, one must define a `class` that extends the `RandObj` trait while initializing it with a `Model`. A `Model` represents a database in which all of the `RandObj`'s random variables and constraints are stored. This `RandObj` will then contain all of the constraints and random variables we will use in our constrained random tests.

There are two main constructs that can be defined inside of a `RandObj`: random variables and constraints.

a) Random variables: These represent random value generators and are associated to constraints. Random variables can either be *regular*, meaning that they can take any value satisfying the constraints, or *cyclic*, meaning that they can not take the same values twice until all values have been covered. Both types are declared using a lower and an upper bound. For example, if we create a `rand(0, 5, Cyclic)`, we will never get the same value twice if we sample it six times, however, on the 7th sampling, the cycle will be reset, and we will start to re-obtain old values.

b) Constraints: Constraints can either be defined alone or in `ConstraintGroup`s. Constraint operators are applied on random variables to create constraints. Conditional constraints may also be defined using the `IfCond` and `ElseC` constructs. All of these constraints can then be enabled and disabled when needed throughout the test suite.

c) Using a RandObj: Once defined, random objects are instantiated and then randomized using the `randomize` method which returns wether or not the constraints were solvable by the CSP solver. The random variables can then be accessed using their respective `value()` methods.

```
1 class Packet extends RandObj(new Model(3)) {
2     val idx = rand(0, 10)
3     val size = rand(1, 100)
4     val len = rand(1, 100)
5     val payload: Array[Rand] =
          Array.tabulate(11)(rand(1, 100))

7     //Example Constraint with operations
8     val s: Constraint = (payload(0) == (len - size))

10    //Example conditional constraint
11    val cond = IfCon(len == 1) {
12       payload.size == 3
13    } ElseC {
14       payload.size == 10
15    }
16    val idxConst = idx < payload.size
17 }
```

Listing 2: Usage of a random object. `rand(min, max, type=Normal)` is used to declare a random variable. Any operation on a random variable generates a constraint.

Listing 2 presents the different ways to define a random variable with constraints. One can define collections of random variables and create constraints on those collections, as was done, for example, in the `payload` random variable. Conditional constraints are shown in the `conditional` random variable, where the constraint depends on the value of the `len` random variable.

Combining constraint-random objects with the provided coverage features enables writing simple coverage-driven randomized tests. However, this may be further optimized by abstracting away groups of wires and operating on an operation or transaction level instead.

C. Verification with Bus Functional Models

Finally, many designers ensure portability and flexibility by equipping their designs with standardized interfaces. The verification engineers can test such components by combining CRV and coverage measures with BFMs to abstract their operation to a transaction level. In this work, we provide an example BFM for AXI4, an open standard by ARM [19].

1) Introduction to AXI4: The Advanced eXtensible Interface (AXI) protocol by ARM is a highly flexible interconnect standard based around five independent channels; three for write operations and two for read operations. Operations, known as transactions, consist of transfers across either set of channels. All channels share a common clock and active-low reset and base their transfers on ready-valid handshaking. The write channels are *Write Address*, *Write Data*, and *Write Response*. The read channels are *Read Address* and *Read Data*.

Consider, for example, a write transaction of 16 data elements in which the manager first provides the transaction attributes (e.g., target address and data size) as a single transfer over the *Write Address* channel followed by the 16 data elements one at a time over the *Write Data* channel. Finally, the subordinate indicates the status of the transaction over the *Write Response* channel. Beware that the write data may be transferred prior to the transaction attributes due to channel independence, and similarly, the *Read Address* and *Read Data* channels may operate independently at the same time [19].

2) Implementation: Our implementation of an AXI4 BFM includes bundles defining the five different channels, abstract classes representing both manager and subordinate entities, transaction-related classes, and the BFM itself, the `FunctionalManager` class. The BFM is parameterized with a DUT that extends a `Subordinate` class and provides a simple, transaction-level interface to control the DUT. As such, its two most important public methods are `createWriteTrx` and `createReadTrx`, which do precisely as their names indicate; create and enqueue write and read transactions.

Internally, the BFM makes use of ChiselTest's multithreading features to allow for (a) non-blocking calls to the methods mentioned above (i.e., one can enqueue multiple transactions without waiting for their completion) and (b) emulating the channel independence more closely. When, for example, a write transaction is enqueued, and no other write transactions are in-flight, the BFM spawns three new threads, one for each required channel. The threads each handle the handshaking necessary to operate the channels.

3) A Test Example: Consider as an example using the BFM to test a module called `Memory`, as shown below. Creating a write transaction with 16 data elements (minimum burst length is 1, hence `len = 15` means a burst of 16 items) takes just one call to a method the majority of whose arguments have default values. It is equally simple to create a subsequent read transaction. Beware that due to channel independence, not waiting for a write to complete before starting to read from the same address may return incorrect results depending on the implementation of the DUT.

```
1 //[...] ChiselTest class delcaration
2     test(new Memory()) { dut =>
3       val bfm = new FunctionalManager(dut)
4       bfm.createWriteTrx(0, Seq.fill(16)
5         (0x7FFFFFFF), len = 15, size = 2)
6       bfm.createReadTrx(0, len = 15, size = 2)
7     }
```

Listing 3: Using the AXI4 BFM with ChiselTest

V. Use Case: Sorting in Hardware

In our research, we received a use case from Microchip [20] in the form of a specification. We implemented and used it to evaluate our verification library.

A. Specification

The provided use case is a hardware implementation of a priority queue, which can be used in real-time systems requiring scheduling capabilities. For instance, timestamps for deadlines can be inserted and sorted such that the host system has access to the closest deadline.

Internally, the hardware priority queue relies on a min-heap tree data structure. The run-times of insertion and removal operations, both having a complexity of $O(\log_k N)$ where N is the number of elements and k the number children per node, are bound by the depth of the tree. By increasing k, the depth of the tree, as well the run-times, can be reduced.

In order to remove elements from the priority queue, a reference ID is needed. Therefore, a reference ID must be added to each element by the priority queue's user.

B. Testing and Verification

The presented CRV and functional coverage functionalities of the ChiselVerify framework were used to verify the modules and the fully assembled queue. Due to the simple interface of the priority queue, which only consists of two boolean flow-control inputs alongside the data fields, only distributional constraints were used to reduce the number of transactions marked as invalid.

The functional coverage report was then used to check how well the inputs were spread over the spectrum of possibilities and to check whether certain input combinations were applied to the DUT at some point throughout the test. As an example, the timed coverage feature made it easy to check whether the `valid` input of the DUT was revoked at some point within 10 clock cycles after issuing an operation by adding the following cross-coverage group:

```
1 cover("timed_valid", dut.io.valid, dut.io.valid)(
2   Eventually(10))(
3     cross("revoked_valid_under_op", 1 to 1, 0 to 0))
```

Listing 4: A timed cover construct.

In order to check whether the DUT matched the specification, a reference model was written for each module. As a reference model for the whole priority queue, a class was written which simulates state and interaction on a transaction/query level. In order to abstract interaction with the DUT, wrapper classes (i.e., classes similar to BFMs) were employed. These make it easy to think on a transaction or operation level when writing tests.

To evaluate the efficiency of ChiselVerify, in terms of lines of code needed, we also verified our sorting hardware with UVM. The main difference, we found, between the two methods was UVM's necessity for boiler-plate classes in order to maintain its reusability. For example, gathering the same functional coverage data using UVM required the following:

- Create a UVM-subscriber based coverage class.
- Instantiate the current DUT (`Heapifier dut = new;`)
- Declare the verification plan:

```
1  covergroup cg_all_zeros_ones;
2  OPS: coverpoint dut.cmd.op {
3    bin insertion = {0};
4    bin removal = {1};
5  }
6  //...
```

- Define `build_phase` and `write` functions.
- Define the coverage class constructor.

The UVM subscriber must then be used inside of a whole UVM test-bench, which itself contains many other classes and constructs. This also holds for UVM's random objects. For comparison, our ChiselVerify test-bench is defined with just 24 lines of code, while its UVM counterpart takes up 96 lines, including only the `uvm_subscriber` class. This is a significant reduction and provides a good indication of how our Scala-based solution minimizes the amount of code needed to utilize advanced verification features.
In summary, for functional coverage, all that needs to be done is to define a verification plan and sample it during a test, and CRV can be done by defining just a random object.

VI. Conclusion

In this paper, we introduced ChiselVerify, an open-source solution that should increase a verification engineer's productivity by following the trend of moving towards a more high-level and software-like ecosystem for hardware design. When using it to test an industry-provided use case, we showed that it requires far less lines of code than UVM, all while obtaining similar results. ChiselVerify's lightweight syntax allows the user to access these helpful tools in a timely manner, thus making it a better fit for agile development than other solutions such as UVM. With this, we brought functional coverage, constrained random verification, and bus functional models to the Chisel/Scala ecosystem, thus improving the current engineer's efficiency and easing the way for software engineers to join the hardware verification world.

Source Access

This work is in open source and hosted at GitHub: https://github.com/chiselverify/chiselverify. We plan also to regularly publish it on Maven.[2]

[2]https://mvnrepository.com/artifact/io.github.chiselverify/chiselverify

REFERENCES

[1] W. J. Dally, Y. Turakhia, and S. Han, "Domain-specific hardware accelerators," *Commun. ACM*, vol. 63, no. 7, pp. 48–57, Jun. 2020.

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

[3] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Commun. ACM*, vol. 62, no. 2, pp. 48–60, Jan. 2019.

[4] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J. Wawrzynek, and K. Asanovic, "Chisel: constructing hardware in a scala embedded language," in *The 49th Annual Design Automation Conference (DAC 2012)*. San Francisco, CA, USA: ACM, June 2012, pp. 1216–1225.

[5] R. Lin. ChiselTest. https://github.com/ucb-bar/chisel-testers2.

[6] C. Spear, *SystemVerilog for verification: a guide to learning the test-bench language features*. Springer Science & Business Media, 2008.

[7] A. S. I. (Accellera), "Universal Verification Methodology (UVM) 1.2 user's guide," https://www.accellera.org/images/downloads/standards/uvm/uvm_users_guide_1.2.pdf, 2015.

[8] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen, "Rfuzz: Coverage-directed fuzz testing of rtl on fpgas," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018, pp. 1–8.

[9] A. Dobis, T. Petersen, and M. Schoeberl, "Towards functional coverage-driven fuzzing for Chisel designs," in *Proceedings of the Fourth Workshop on Open-Source EDA Technology (WOSET)*, 2021.

[10] T. Alcorn, "Chisel formal verification," https://github.com/tdb-alcorn/chisel-formal, accessed: 2021-06-03.

[11] R. Nilsson, *ScalaCheck: The Definitive Guide*. Artima Inc, 2014.

[12] K. Laeufer, "Chisel formal verification extension," https://github.com/ekiwi/kiwi-formal, accessed: 2021-06-03.

[13] D. Kasza, "Chisel formal verification extension," https://github.com/danielkasza/dank-formal, accessed: 2021-06-03.

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

[15] A. B. Mehta, *Constrained Random Verification (CRV)*. Cham: Springer International Publishing, 2018, pp. 65–74. [Online]. Available: https://doi.org/10.1007/978-3-319-59418-7_5

[16] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach, "Reusability is firrtl ground: Hardware construction languages, compiler frameworks, and transformations," in *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2017, pp. 209–216.

[17] Veripool, "Verilator," https://www.veripool.org/wiki/verilator.

[18] K. Kuchcinski and R. Szymanek, "Jacop - java constraint programming solver," 2013, cP Solvers: Modeling, Applications, Integration, and Standardization, co-located with the 19th International Conference on Principles and Practice of Constraint Programming ; Conference date: 16-09-2013.

[19] ARM, "Amba axi and ace protocol specification axi3, axi4, and axi4-lite ace and ace-lite," https://developer.arm.com/documentation/ihi0022/e/.

[20] "Microchip," https://www.microchip.com/, accessed: 2021-08-29.