Fuzzing ROHD Hardware Designs in Dart

CHRISTIE ELLKS, University of California, Los Angeles

This capstone project aims to automatically generate bug revealing inputs for hardware designed in Dart using Intel's Rapid Open-source Hardware Development (ROHD) Framework. ROHD and the ROHD-Verification Framework (VF) provides users with the benefits of developing and testing in the software domain while implementing hardware. Among these benefits is the potential to leverage software fuzzing tools that are typically cumbersome to apply to hardware development. This project implements a methodology of preparing a ROHD-VF testbench for fuzzing and explores fuzzing tools for the Dart language. Ultimately, an enhanced version of an existing coverage guided Dart fuzzer called Dust is integrated with the extended ROHD-VF testbench to achieve hardware fuzzing in software.

1 INTRODUCTION

Hardware verification will always be a critical and primary aspect of hardware development. Undetected bugs and security vulnerabilities in hardware can be catastrophic and extremely costly. Since patches are not always remotely deployable as in software development, in some cases public recalls and/or releasing new hardware may be the only remedy for the situation. To avoid this, a large portion of hardware implementation is dedicated to the verification phase. In fact, an average of 51% of FPGA project time was dedicated to verification in 2020 [12]. To make this step more efficient, developers have begun applying software fuzzing techniques to hardware testing [11]. Software fuzzing is a broad field of automated testing which aims to reveal flaws in software programs by generating inputs that cause crashes and/or explore unique execution paths through the program.

Discovering crashing inputs for hardware designs is a complicated challenge, especially as hardware becomes increasingly complex. Previous work has attempted to bring fuzzing into the hardware domain by creating new hardware fuzzers, however this has not resulted in overwhelming success [11]. An alternative approach implemented by the Hardware Fuzzing Pipeline (HWFP) (Trippel et al., 2021) seeks to take advantage of the already existing software fuzzing tools by bringing the hardware design and testbench into the software domain. Trippel et al. (2021) introduced the various challenges of adapting hardware into software and developed a harness which mollifies these complications. However, an alternative solution is to design the hardware in software from the beginning, verify the software version, then convert to hardware.

ROHD, spearheaded by Max Korbel at Intel, was created as an alternative to developing hardware using an HDL like VHDL and SystemVerilog [8]. This tool addresses the question of how to develop hardware in the software domain. ROHD is a Dart framework that exercises the benefits of software programming in an easy to learn language with straightforward testing capabilities. Additionally, ROHD offers ROHD-VF (Verification Framework), a secondary framework that provides a streamlined approach to generating and running testbenches for designs constructed with ROHD [9]. Furthermore, users can generate their designs in SystemVerilog after developing and testing them using ROHD and ROHD-VF in the Dart language.

This capstone project applies fuzzing techniques to ROHD-VF testbenches to improve the hardware verification process for designs implemented via ROHD. To integrate ROHD-VF testbenches with a fuzzer, this project first develops a methodology of formatting fuzzer generated inputs and feeding them into the design under test (DUT). The second task is to find a suitable fuzzer for the Dart language by exploring two fuzzing approaches and tools: blackbox fuzzing with libFuzzer [10] and coverage-guided, greybox fuzzing with Dust [6]. The latter approach was only viable after fixing the Dust source code to make it executable and customizing it to fit this application's specifications. Finally, integration testing of two ROHD-VF testbenches with the enhanced Dust fuzzer was conducted.

This report is organized as follows:

- **Section 2** provides background knowledge of ROHD-VF necessary for this project.
- **Section 3** examines the related work of the HWFP and compares it to the current project.
- **Section 4** outlines the work completed for this project, partitioning it into subsections regarding preparing ROHD-VF testbenches, blackbox fuzzing, and finally greybox fuzzing.
- **Section 5** describes the integration testing performed on two hardware designs implemented in Dust via ROHD.
- **Section 6** discusses lessons learned throughout this project and future directions for this work.

2 BACKGROUND

2.1 ROHD-VF

The ROHD-VF methodology, depicted in Figure 1, outlines a concise template to create testbenches for ROHD designs. Testbenches are constructed by extending ROHD-VF classes to create custom components similar to Universal Verification Methodology (UVM) components. This section will provide a simplified overview of a testbench, omitting descriptions of the components not essential to this project. A single `SequenceItem` component contains the set of values to transmit to DUT input ports. A `Sequence` contains a set of `SequenceItems` and specifies the order those `SequenceItems` should be sent to the DUT.

A `Sequencer` component takes a `Sequence` and designates how the `SequenceItems` are forwarded to the `Driver`. The `Driver` component is responsible for injecting the inputs from the `Sequencer` into the DUT's interface. A `Monitor` component acts as a listener for the DUT's input and output ports and triggers events for other components like logic checking in the `Scoreboard` component. The `Scoreboard` implement the testcases and/or specify invariants for the DUT by comparing values passed from the `Monitor` to expected values.

For this project, we make the following assumptions:

- `SequenceItems` correspond to a set of values, one per input port of the DUT, to be transmitted at a single in time.

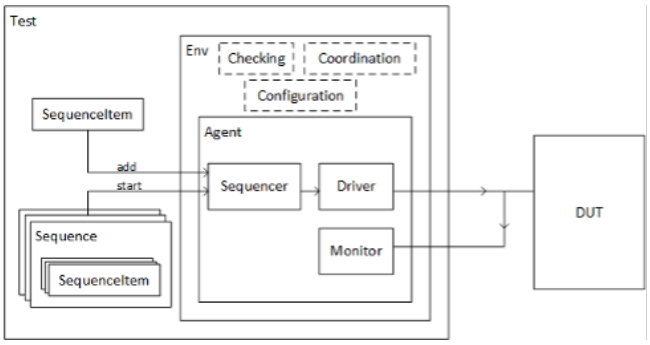

Fig. 1. Overview of ROHD-VF workflow.

- The only function of the `Sequence` component is to order the `SequenceItems`.
- The `Sequencer` component simply forwards the `Sequences` directly to the `Driver`.
- The user is responsible for the `Driver`'s setup functionality, manually injects all clock and reset signals for the DUT, and configures the run() function which specifies when to send the next `SequenceItem` using the drive() function.
- The `Driver` injects all values in a `SequenceItem` onto the DUT's interface when the drive() function is called.

To integrate a ROHD-VF testbench with a fuzzer, we focus on converting fuzzer generated inputs into `SequenceItems` which are eventually fed into the DUT. Given the assumptions above, the fuzzer can only specify the values of the input ports, excluding the clock and reset signals. This means the fuzzer cannot trigger resets or simulate functionality under clock malfunctions at this time.

3 RELATED WORK

3.1 Fuzzing Hardware in Software Domain

In general, coverage measures how much of a program is executed with a given input. Edge, or branch, coverage in particular measures the executed branches (or paths) through a program out of the total possible paths. Trippel et al. (2021) specify that edge coverage is more suitable for fuzzing hardware than other coverage metrics because it corresponds with tracking the various states of the hardware throughout the test duration. Their Hardware Fuzzing Pipeline (HWFP) uses a tool called Verilog to compile a SystemVerilog RTL design and its testbench into a C++ Hardware Simulation Binary (HSB). It then uses their custom harness to feed inputs from C/C++ fuzzers, including libFuzzer and AFL, to the HSB using TL-UL bus transactions. Experiments demonstrated that instrumenting solely the DUT within the HSB, as opposed to instrumenting both DUT and testbench, improved the fuzzer's performance. Additionally, Trippel et al. (2021) notes an important distinction of fuzzing hardware versus software is the notion that fuzzed inputs must be mapped to simulate a series of input signals over time. Their harness originally achieved this by continually looping over the DUT's input ports and passing in data from the fuzzed input stream corresponding to the port width, then progressing the clock by one tick. However, ultimately Trippel et al. (2021) decided on the bus level interface and constructed a "bus-centric grammar" to ensure the fuzzers generated syntactically correct inputs for their harness.

This project uses Trippel et al. (2021) 's work as a guideline for fuzzing hardware in the software domain. We continue to rely on edge coverage for fuzzing campaigns and adapt many of the ideas used in the HWFP for designs written using ROHD. The ROHD-VF testbench combined with the ROHD design parallels the HSB fuzzed in the HWFP. To map fuzzed inputs over a time domain, we split the input data stream into independent `SequenceItems`. A key difference here is that the HWFP original harness fed one new input per clock tick whereas this project injects one full `SequenceItem` (one signal per DUT input port) within one clock tick. Furthermore, using ROHD-VF avoids the need to feed inputs at the bus interface level since we have programmatic access to the DUT's interface. To ensure the fuzzed inputs will be compatible with the DUT, this project assumes the fuzzer produces binary strings and augments the testbenches to parse the binary strings into valid `SequenceItems`.

4 METHODS

4.1 ROHD-VF Fuzzing

From the perspective of the ROHD-VF testbench, it is assumed that a binary string of variable length is passed in through main() from a fuzzer.

This project defines a custom `DUTSeqItem` class that extends ROHD-VF's `SequenceItem`. The fuzzed inputs must be converted into these `DUTSeqItems` to be injected into the DUT's interface. The `DUTSeqItem` class has the following properties:

- An int type member variable for each input port specified by the DUT's `Interface`, named by the corresponding port's name.
- A Map<String, int> type member variable called ports where keys are input port names and values are the value to be injected to the specified port.
- A default constructor that accepts a list of integers and:
  (1) sets each int type member variable to a value from the input list and
  (2) adds the port name – value pair to the ports map.

To reduce the user workload, I have created a separate Dart script that automatically generates this `DUTSeqItem` class for a given ROHD Interface and writes it to a separate Dart file. The user must run this script and import the generated file at the top of their testbench prior to initiating any fuzzing campaigns.

To format the fuzzed inputs, the provided binary string must be segmented into a list of integer lists which corresponds to a single `Sequence` component. Each integer list corresponds to one `DUTSeqItem` and has a length equal to the number of input ports of the DUT. The bit width of each input port corresponds to the number of bits from the binary string used as the injected value for that port. We implement this by accessing the DUT's Interface as shown in Algo. 1. (Note that by default, Dart Maps are ordered upon key insertion. Therefore the returned interface ports map in this algorithm and in the construction of the `DUTSeqItem` will correspond to the same order of ports.) One drawback of this technique is the possibility that not all bits in the fuzzed binary string will be used in the DUT. For example, if the binary string is 10 bits long, but the

Algorithm 1 Segment Fuzzed Binary Input String

Require: *input* is a bit string
1: $n \leftarrow input$
2: $sumPortWidths \leftarrow$ sum of all input port widths
3: $sequence \leftarrow []$
4: **while** $n.length >= sumPortWidths$ **do**
5: $seqItem \leftarrow []$
6: **for** $port \in dut.ports()$ **do**
7: **if** $port.isInput$ **then**
8: $portValue \leftarrow$ toInt(n[: $port.width$], radix=2)
9: $seqItem$.append($portValue$)
10: $n \leftarrow n[port.width :]$
11: **end if**
12: **end for**
13: $sequence$.append($seqItem$)
14: **end while**

total number of bits used by the DUT's input ports is 4, then the segmentation will form two lists and disregard the remaining 2 bits. This can potentially be remedied by precomputing the sum of all input port bit widths and constraining the fuzzer generated inputs to have lengths that are multiples of that sum.

Finally, the user must specify invariants in their customized `Scoreboard` component within the testbench. All invariant violations must be set to throw an error so that the fuzzer can detect it. The rest of the testbench implementation is up to the user to follow the guide provided by ROHD-VF. With the few extra steps outlined in this section, the testbench is ready to be fuzzed.

4.2 Blackbox Fuzzing

Due to lack of readily available Dart fuzzers, an initial blackbox approach was taken as a proof of concept that fuzzing would work with the testbench setup described above. Dart provides support for compiling Dart source code into a native executable which can be called from C/C++ programs. The C/C++ coverage guided fuzzer, libFuzzer, was then treated as a blackbox fuzzer since it could not instrument the executable. The fuzz target required by libFuzzer was implemented to call the Dart executable and throw an error if anything is written to stderr by the blackbox program. This catches errors that occur in the Dart program and saves the crashing inputs to the corpus. To sanitize the inputs, the byte stream supplied by libFuzzer was converted into a binary string prior to feeding it to the executable. As anticipated, this blackbox fuzzing approach did not perform well as it had no way to measure the coverage of the inputs within the executable. However, providing crashing input seeds did successfully detect errors in the executable. Therefore this approach demonstrated that the underlying ideas of this project were viable given the existence of an adequate Dart fuzzer.

4.3 Greybox Fuzzing with Dust

Since the blackbox approach was not a practical solution, the next step was to revisit the only Dart greybox fuzzing option, Dust. Dust is a "coverage-guided fuzz tester for Dart inspired by libFuzzer and AFL", however there were bugs in the published source code that made it initially impossible to run. Therefore a significant amount of time was devoted to analyzing Dust's code to track down the source of the errors. (For details of the uncovered bugs, see Section 6.1.1). Once the bugs were resolved, Dust was tested on a handful of testcases to ensure it successfully fuzzed simple Dart code and could generate a corpus of failing inputs.

For this project, the fuzzed inputs are restricted to be binary strings. To guarantee the fuzzed inputs adhere to this requirement, Dust's mutator implementation was modified to accept an additional flag specifying valid characters for the generated string inputs. Although Dust already had a built-in support for supplying custom mutator scripts, it was not ideal due to the custom scripts only providing a single string to mutate. The script cannot access the existing corpus to implement more intricate mutations such as the crossing over or splicing of two existing seeds to generate a new one. Given the format of the original source code, a user could not enable crossing over / splicing while restricting valid characters. Therefore Dust's source code for `Mutator` objects was directly edited to support the specification of valid input characters.

Upon the inspection of the Dust source code, it was revealed that Dust measures line coverage as opposed to the desired edge coverage. (The documentation did not specify the coverage type.) To remedy this, attempts were made to elicit branch coverage information from the VmServices spawned each time a new fuzzed input is passed to the testbench. However, branch coverage metrics are a new feature for the Dart language and support for reading branch coverage from a VmService object was only implemented in January of 2022 [1]. This caused many issues with attempting to pull in branch metrics from the children processes. Therefore, as a compromise, a pseudo edge-type metric was developed by saving the full execution path as if it were a unique, single line of the program. This promotes that addition of inputs that reveal new execution paths to the corpus whereas prior it might have been overlooked if it did not execute any novel lines. This method does not compute edge coverage since it cannot determine the total number of possible paths in the program without more support from Dart libraries. However, it does provide more insight to the hardware execution path than the original Dust implementation.

5 RESULTS

Successful integration testing of ROHD-VF testbenches and the enhanced Dust was performed on two designs.

5.1 DUT: Simple Counter Module

The simpler of the two designs is a `Counter` module that increments the output value `val` at each clock cycle where the input signal `enable` is high. Therefore the DUT's main invariant is that `val` is expected to increase by 1 if `enable == 1` and `reset != 0`. A bug was manually inserted into this `Counter` DUT that caused `val` to be set to 3 if in the previous clock cycle it had been greater than or equal to 4. The `Scoreboard` component in the testbench was implemented to throw an error if the stated invariant was violated. The automatically generated `DUTSeqItem` component based on the `CounterInterface` has a constructor that takes a List<int> of length 1 and uses it to set its one member variable, `enable`.

Running the enhanced Dust fuzzer successfully produced the failing input of '0101110110' after 2.5min of fuzzing when no seeds were provided. (Dust's default seed is an empty string when no seeds are passed from the invoking terminal command.) This string has a formatted input of [[0], [1], [0], [1], [1], [1], [0], [1], [1], [0]] which corresponds to ten `DUTSeqItems`. The `Driver` automatically uses each of these `DUTSeqItems` to inject its member variable `enable` value into the `enable` input port of the DUT at each negative edge of the clock signal. After the sixth `DUTSeqItem` is injected, the DUT's `val.value` is 4 since `enable` has only been high for four clock cycles. Therefore after the eighth `DUTSeqItem` is injected, the testbench fails since `val.value` is set back to 3 instead of incrementing to 5 due to the injected bug. (Note the testbench does not fail after the seventh `DUTSeqItem` since enable was low.) When fuzzing with initial seeds of '111' and '000', the fuzzer generated the failing input of '1111111' in a matter of seconds since it did not have to start with only the empty string in the initial corpus.

This example demonstrates the ability for the testbench to successfully parse the fuzzed inputs into `DUTSeqItems` and feed them to the DUT. It also shows that the fuzzer can detect failures in a design that keeps state over time based on the implementation of the testbench's `Scoreboard` component. In this case, `Scoreboard` stored the previous `val.value` and used it to compute the expected `val.value` for the next clock cycle to verify the invariant.

5.2 DUT: Two Input Adder Module

The second DUT was an `Adder` module that takes two 4-bit inputs, `in1` and `in2`, and passes their 5-bit sum onto the output bus `out`. The module computes the sum in a bitwise manner with maintaining 'carries' for each bit. In this design, a bug was implemented to miscalculate the final carry bit when the most significant bit (MSB) of 'in1' == 0 and the carry from the previous bits equals 1. For example, (`in1`: 4'b0111) + (`in2`: 4'b1001) would output 5'b00000 even though 5'b10000 is expected due to the miscalculated final carry bit. For this module, the automatically generated `DUTSeqItem` class has a constructor that takes a List<int> of length 2 since there are two inputs for the `AdderInterface` and stores the values to its two member variables. The fuzzed input formatter reads in four bits per input port from the fuzzed binary string.

The enhanced Dust fuzzer successfully generated the failing string input of '01111111111111111111' which is formatted as [[7, 15], [15, 15]], corresponding to two `DUTSeqItems`. In this case, after injecting the first `DUTSeqItem`, the testbench saw (`in1`: 4'b0111) + (`in2`: 4'b1111) -> (`out`: 5'b00110) when it was expecting (`out`: 5'b10110). This demonstrated that the formatter was able to convert the fuzzed input into `DUTSeqItems` when the DUT's interface had multiple input ports with bit widths larger than 1 (which the previous DUT did not test). Additionally, the fuzzer was able to generate an input for a specific error case. In the previous example, any input that contained more than four 1's would have produced the error. However, for this bug the input specifically needed a 0 as the MSB of `in1` and the sum of the previous bits needed to result in a carry. The fuzzer found such an input starting with no seeds in 18 minutes.

6 DISCUSSION

6.1 Lessons Learned

This project required the learning of many new tools, including the programming language Dart. Thankfully, Dart is a very intuitive language and was simple enough to pick up quickly. Its package manager, Pub, was straightforward as well. ROHD and ROHD-VF were similar enough to SystemVerilog that the concepts translated over quite smoothly, making this project feasible. Debugging and extending Dust involved learning about more specific, technical aspects of Dart including Isolate spawning (Dart's version of subprocesses) and coverage metrics. Furthermore, it improved my skills in tracing through existing code authored by someone else.

6.1.1 Debugging Dust. Attempting the original Dust's provided toy example from the documentation resulted in a cryptic error message about an observatory not starting on a given port number. An active issue from another user experiencing the same observatory error indicated this was an issue with the tool itself and not an error with local setup. Ultimately, the initial error was determined to be caused by an outdated, hardcoded string path to a script. Fixing this uncovered a latent bug which was only triggered if the fuzzing target prints to stdout. Because Dust uses Isolates to spawn new processes, it relies on stdout to communicate back to the main thread. If the fuzzing target prints to stdout, the main thread is not able to use the JSON decode function successfully on the input stream from stdout. (Dust's documentation had no mention to prevent the fuzz target from printing). This was amended in the source code by indicating the end of the process's execution in stdout and had the main thread only decode the JSON output that came after the end of the process's prints. Adding this check allowed the fuzzer to run as expected and it generated some failing inputs for simple toy examples.

6.2 Future Work

There are numerous improvements and potential extensions for this project, starting with additional testing. The current workflow can be tested with larger, more complex ROHD designs to see how the Dust fuzzer performs. Currently Dust instruments both the testbench and DUT; it would be interesting to see if instrumenting only the DUT provides performance enhancements as it did in the HWFP [11].

Future work could be done to the fuzzed input parsing to apply those inputs to the Sequence and Driver functionalities and/or integrate fuzzing with more complex components than the simple version tested with during this project. The Scoreboard class could also be extended further to indicate in the failure message at what point in the Sequence the error occurred. To automate as much as possible for users, it would be great to programmatically generate a majority of the ROHD-VF testbench based on a provided DUT interface. Then the user would simply fill in the invariants in the Scoreboard class and indicate when Dart errors should be thrown.

ROHD-VF testbenches can be applied to hardware designs that make calls to SystemVerilog module. It would be interesting to see how the fuzzer handles these hybrid designs. Finally, more work can be done on top of the improvements this project made to Dust. For example, more complex mutation functions such as collision

crossovers [7] could be employed to potentially improve the search for new crashing inputs. This would either entail directly modifying the source code or extending the source to allow custom mutator scripts to access the seed corpus. For this project's specific use case, it would be beneficial to be able to specify valid input lengths for generated strings (in our case valid lengths would be multiples of the sum of the DUT's port widths).

7 CONCLUSION

This capstone project lays the groundwork for fuzzing hardware designs implemented in Dart with the ROHD framework. It outlines the step necessary to prepare a ROHD-VF testbench for fuzzing and has made application specific enhancements to Dust. It provides proof of concept with small, toy examples and demonstrates a working example of fuzzing a ROHD design of a Counter DUT. This project has demonstrated that fuzzing the ROHD designs can generate failing inputs, revealing flaws in the hardware design. Now that this foundation has been laid, more work can be done to further improve the verification phase of ROHD designs through fuzzing.

8 APPENDIX

Hyperlinked Repositories with Setup and Execution Documentation:

Hardware Fuzzing in Dart [5]
Enhanced Dust [4]
Counter DUT Example [3]
Blackbox Approach [2]

REFERENCES

[1] Liam Appelbe. 2022. Adding branch coverage RPC to source report. https://github.com/dart-lang/sdk/commit/9d79a890f393b33abf2117f4cb239b90d34a6dff
[2] Christie Ellks. 2022. Blackbox Approach. https://github.com/clincoln8/rohd-fuzzing/tree/main/deprecated/blackbox-fuzz
[3] Christie Ellks. 2022. Counter DUT Example. https://github.com/clincoln8/rohd-fuzzing/tree/main/examples/counter
[4] Christie Ellks. 2022. Enhanced Dust. https://github.com/clincoln8/dust
[5] Christie Ellks. 2022. Hardware Fuzzing in Dart. https://github.com/clincoln8/rohd-fuzzing
[6] Michael R Fairhurst. 2019. Dust. https://github.com/MichaelRFairhurst/dust
[7] Ahmad BA Hassanat and Esra'a Alkafaween. 2017. On enhancing genetic algorithms using new crossovers. *International Journal of Computer Applications in Technology* 55, 3 (2017), 202–212.
[8] Max Korbel. 2021. Rapid Open-source Hardware Development. https://github.com/intel/rohd
[9] Max Korbel. 2021. ROHD-VF. https://github.com/intel/rohd-vf
[10] K. Serebryany. 2021. libFuzzer. https://llvm.org/docs/LibFuzzer.html
[11] Timothy Trippel, Kang G Shin, Alex Chernyakhovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks. 2021. Fuzzing Hardware Like Software. *arXiv preprint arXiv:2102.02308* (2021).
[12] Hasini Witharana, Yangdi Lyu, Subodha Charles, and Prabhat Mishra. 2022. A Survey on Assertion-Based Hardware Verification. *ACM Comput. Surv.* (Jan 2022). https://doi.org/10.1145/3510578 Just Accepted.