Research Project Proposal
Increasing Efficiency and Explainability of Hardware Verification with Fuzzing Techniques

Eric Ackermann, CISPA

July 17, 2024

Hardware Description Languages (HDLs) such as SystemVerilog allow engineers to rapidly design and prototype electronic circuits. While originally intended only for simulation and verification of designs, modern synthesis tools can synthesize HDL to Field Programmable Gate Array (FPGA) configurations or even standard-cell Application-Specific Integrated Circuits (ASICs). For both FPGAs and ASICs, the options for debugging a design in hardware are fairly limited. At the same time, especially for ASICs, changes after tape-out (manufacturing of the first ASIC) are prohibitively expensive. Thus, functional verification (i.e., testing of the design before synthesis) is crucial for avoiding bugs to propagate into the actual hardware.

However, existing methodologies commonly rely on constrained-random generation of test cases only. Thereby, computational effort of verification is high, and there is no guarantee that all valid test cases will be generated during the verification run time. Anecdotal evidence also suggests that constrained-random solvers tend to only cover a subset of the valid input space under certain conditions.

For the purpose of functional verification, verification languages such as SystemVerilog include language constructs for user-defined input constraints and collecting functional coverage. Thus, this project aims at exploiting these features to enable a fuzzing approach for the generation of test cases, aiming at faster completion of functional verification. To this end, the existing input constraints can serve as corpus for the fuzzer, while functional coverage can serve as feedback.

Background

SystemVerilog is a Hardware Description Language (HDL) consisting of a synthesizable and a non-synthesizable subset. The synthesizable subset is used to model hardware circuits and can be synthesized onto Field-Programmable Gate Arrays (FPGAs) or Application-Specific Integrated Circuits (ASICs), allowing the user to program hardware circuits in the same way as software programs. Hardware verification is crucial for preventing bugs in the synthesizable code (Device under Test, DUT) from reaching tape-out (production of an ASIC), which would cause expensive re-engineering of the circuit. Thus, the extensive non-synthesizable subset of the language consists of object-oriented constructs that are intended primarily for hardware verification [6]. Thereby, object-oriented patterns such as encapsulation of data in classes, factory methods, inheritance and polymorphism can be used in verification, increasing programmer efficiency. Specialized constructs such as interfaces allow bridging the gap between the synthesizable circuit and the non-synthesizable verification testbench.

Universal Verification Methodology (UVM) is an industry-standard verification methodology for hardware designs, built on top of SystemVerilog and a testing framework with the same name [5]. At the core of UVM, a transaction object contains a transaction for a device (e.g., in a DMA device, the starting address, destination address and length of a transfer). The transaction is handed to an agent object, which is responsible for breaking the transaction down into individual bus transactions and transferring them to the driver. The driver uses an interface object to pass the input transaction to the DUT, usually via a standard bus such as AXI. A monitor analyzes the DUT's output and generates a response transaction (e.g., a memory request that the DMA device actually performed). A scoreboard computes an expected response from the input transaction. Finally, a checker compares the real and expected response and raises an error if the device did something unexpected. In addition, there are SystemVerilog assertions which can be used to raise errors when invariants are violated. Such invariants are, e.g., typically defined in the specifications of bus protocols.

Crucially, in UVM, transactions are *generated randomly* by a generator object. To this end, a constraint language is built into the SystemVerilog language. This allows the verification engineer to define the properties of valid transactions. For example, a DMA might only accept addresses that are 8-byte aligned, or the destination address might have to be either a device register or a memory address depending on the transaction type. Especially for components such as bus infrastructure components, constraints can be derived directly from the bus protocol specification and possibly shared between projects. Thus, the constraints essentially define a *corpus* of valid inputs. The methodology assumes that everything not covered by random transactions *cannot* be encountered at run time (e.g., inputs might only be possible if a different component violates the bus protocol).

A crucial question in hardware verification is: When has verification been completed? To this end, SystemVerilog includes a functional coverage system [6]. Functional coverage allows users to indicate which states of the DUT and testbench should be monitored (e.g., state machines, input transactions, output transactions, etc.) and when they can be sampled (e.g., on a clock edge). Further, a system of coverage bins allows precise control over the expected outputs. For example, bins can ensure that all possible types of transfer in a DMA device have been completed or that addresses corresponding to all memory regions and alignments have been covered. Goal of the verification process is to reach 100% coverage [5].

Alternatives to UVM include, e.g., formal verification techniques. However, while UVM can be used with all industry-standard SystemVerilog simulators (including the free Xsim from Xilinx), formal verification requires expensive additional tooling and training, making UVM attractive especially for smaller companies.

Proposed Project

While working on the verification testbench of the research hardware component Northcape [1], which is a novel byte-granular memory management unit, the author noticed that UVM has two major practical shortcomings. First, due to the complexity of the design and testbench, only between 10 and 100 tests per second and CPU core are completed for several components in the system. Second, due to poor distributions of the randomized transactions, certain scenarios were never tested even after many transactions. For example, the transaction has a boolean flag that distinguishes between valid and an invalid bus transactions. For a valid bus transaction, a number of constraints on the input data needs to be satisfied (for example, the access cannot over- or underflow the corresponding buffer in memory) and a status code indicating success needs to be returned. For an invalid bus transaction, the transaction needs to violate at least one constraint for a valid transaction and an error needs to be indicated. In over 50,000 automatic tests, not a single invalid transaction was generated. The author has no way of knowing whether there is an error in the constraints, causing the constraint solver to only be able to generate valid transactions, or if he has simply not yet run enough test cases. Either way, while the distribution of valid and invalid test cases should be close to 50%, in practice it is heavily skewed towards valid transactions.

Thus, this project aims at addressing these two shortcomings of constrained-random testing using a fuzzing approach. The envisioned contribution of the project is achieving 100% functional coverage with fewer tests than purely random generation. Also, the author imagines that it should be possible to determine whether it is possible to reach 100% coverage using the provided constraints. Crucially, the input constraints already built into the language and verification methodology can be used to generate a corpus of valid inputs for the fuzzer. The fuzzer can use functional coverage as feedback. Instead of crashes, the fuzzer has found a bug when the checker indicates abnormal behavior of the DUT, when an assumption is violated or a timeout occurs (e.g., when the DUT locked up). Thereby, the proposed system can be used as a drop-in replacement for constrained-random testing, such that the existing methodology and test benches can be reused.

The implementation of the approach can be built on top of the open-source simulator Verilator, which supports most language constructs of SystemVerilog. There is ongoing work targeting support of constrained-random testing in Verilator[1], on top of which the proposed system could be implemented and evaluated. Alternatively, the proposed system could be built as plugin for Xilinx Xsim or a different commercial simulator.

The author is willing to offer Northcape and its UVM testbench as a DUT for evaluation of the approach. Alternatively, open source projects such as the CVA6 RISC-V CPU[2] include a UVM testbench and can be used as a DUT.

Scope of the Project and Requirements

The author would like to emphasize that the proposed project is not primarily a hardware project, but mostly a testing and software engineering project. It might be a good fit especially for engineers with an interest in fuzzing and verification. The author believes that this project might make a good master thesis. The author also believes that there is the potential of building a product based on this approach, under the assumption that inclusion of the fuzzer into a commercial-grade simulator like Xsim is possible.

State of the Art

The idea of generating new test cases based on coverage is not entirely new. However, to the author's knowledge, no paper has considered simply replacing the constrained-random generation of transactions on its own, while keeping the rest of the UVM methodology in place. Instead, existing literature always tries to solve multiple problems at once, e.g., generating constraints from scratch or using a different coverage metric than human-generated functional coverage. Also, the explainability of (non) covered cases seems to be understudied.

Trippel et al. use Verilator to generate a software model for a hardware component and use AFL for fuzzing the *software model* [8]. They use edge coverage (in software) as sole coverage metric and exclusively rely on assertions to detect errors of the DUT, possibly missing complex bugs that cannot be modeled (or were forgotten to be modeled) as assertion but would be caught by scoreboard and checker.

Dobis, Petersen, and Schoeberl implement a scheme similar to the proposed one, but only targeting the Chisel HDL. Also, instead of relying on the existing constraints in the testbench, they assume the existence of input files with exemplary valid inputs for the generation of transactions in the fuzzer [2].

Laeufer et al. run their verification on FPGAs to decrease run time. They also generate test suites with functional coverage models automatically instead of relying on human-generated models, possibly missing domain-specific edge cases [4].

Guzey and Wang focus on automatic generation of input constraints, based on a functional coverage model, possibly generating inputs that the DUT needs not accept because they are invalid [3].

[1] `https://antmicro.com/blog/2024/03/introducing-constrained-randomization-in-verilator/`
[2] `https://github.com/openhwgroup/cva6`

Finally, Teplitsky, Metodi, and Azaria pursue the same idea of using existing constraints and functional coverage. They aim at achieving 100% coverage faster by carefully manipulating the output distributions of the constrained-random generator, but continue relying on constrained-random test generation in principle [7].

Bibliography

[1] Eric Ackermann, Noah Mauthe, and Sven Bugiel. "Work-in-Progress: Northcape: Embedded Real-Time Capability-Based Addressing". In: *2024 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. 2024.

[2] Amelia Dobis, Tjark Petersen, and Martin Schoeberl. "Towards Functional Coverage-Driven Fuzzing for Chisel Designs". In: Workshop on Open-Source EDA Technology (WOSET 2021). Technical University of Denmark, Nov. 4, 2021. DOI: `10.3929/ethz-b-000539444`. URL: `https://www.research-collection.ethz.ch/handle/20.500.11850/539444` (visited on 05/28/2024).

[3] Onur Guzey and Li-C. Wang. "Coverage-Directed Test Generation through Automatic Constraint Extraction". In: *2007 IEEE International High Level Design Validation and Test Workshop*. 2007 IEEE International High Level Design Validation and Test Workshop. Irvine, CA, USA: IEEE, 2007, pp. 151–158. ISBN: 978-1-4244-1480-2. DOI: `10.1109/HLDVT.2007.4392805`. URL: `http://ieeexplore.ieee.org/document/4392805/` (visited on 07/15/2024).

[4] Kevin Laeufer et al. "RFUZZ: Coverage-Directed Fuzz Testing of RTL on FPGAs". In: *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). Nov. 2018, pp. 1–8. DOI: `10.1145/3240765.3240842`. URL: `https://ieeexplore.ieee.org/abstract/document/8587711` (visited on 05/28/2024).

[5] Ray Salemi. *The UVM Primer: A Introduction to the Universal Verification Methodology*. Breinigsville, PA: Ray Salemi, 2013. 190 pp. ISBN: 978-0-9741649-3-9.

[6] Chris Spear. *SystemVerilog for Verification: A Guide to Learning the Testbench Language Features*. Springer Science & Business Media, Apr. 22, 2008. 455 pp. ISBN: 978-0-387-76530-3.

[7] Marat Teplitsky, Amit Metodi, and Raz Azaria. "Coverage Driven Distribution of Constrained Random Stimuli". In: 2015. URL: `https://www.semanticscholar.org/paper/Coverage-Driven-Distribution-of-Constrained-Random-Teplitsky-Metodi/d0eaf654fe6f4f80d2f9ddd20083eca037048d7b` (visited on 07/15/2024).

[8] Timothy Trippel et al. "Fuzzing Hardware Like Software". In: 31st USENIX Security Symposium (USENIX Security 22). 2022, pp. 3237–3254. ISBN: 978-1-939133-31-1. URL: `https://www.usenix.org/conference/usenixsecurity22/presentation/trippel` (visited on 05/28/2024).