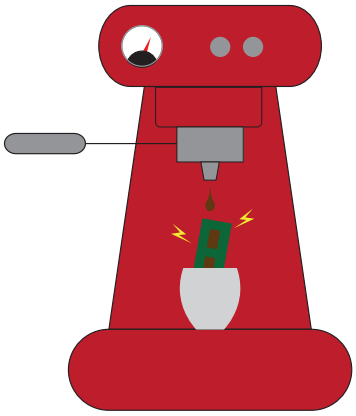


Verification of digital circuits using Java

02125 Bachelor Project

by
Rasmus Wiuff **s163977** s163977@dtu.dk
github.com/rwiuff/SteelBrew

Supervisor
Martin Schoeberl masca@dtu.dk

Draft

Abstract

Add abstract

TODO LIST

CONTENTS

1 Summary

This section summarises the sections of the report.
Section 2 surmises UVM and some chosen chip verification frameworks, pivoting towards the motivation for, and introduction of, this project.
Section 3 breaks down the project into challenges and discusses solutions to these challenges.
Section 4 goes through the design considerations along with diagrams outlining the projects intended structure.
Section 5 describes the implementation and shows solutions of interest.
Section 6 explains the testing used to verify the project.
Section 7 summarises the project and discusses the results.

2 Introduction

Related works

In the world of designing integrated systems on chips, it is crucial to describe these in some hardware description language in order to test designs before actual manufacturing. Part of testing these designs include writing tests in some framework supporting some abstraction and then simulate the chip using the applied framework, a process known as *verification*. One verifies the *Device Under Test* or DUT. One of the most common methodologies is *Universal Verification Methodology*, or simply UVM.

2.1. The UVM methodology

Using UVM, one have to built testbench components. These include e.g. drivers for converting tests into proper DUT stimulus, monitors for reading the state of the DUT and scoreboards for comparing expected behaviour to actual behaviour, etc. UVM has the great benefit, that once these components have been defined, they can be reused within the scope of some system of designs. This means a high overhead, with high reusability.

2.2. Current verification tools

In today's landscape a myriad of verification tools exists. Projects like Chisel[1], and it's forked project SpinalHDL[2] discards the traditional hardware description languages and testbenches altogether and implements their own, and pyuvm[3] takes UVM into python using cocotb[4] as a backend driver. These tools do however have some problems. Chisel and SpinalHDL requires developers to learn a new description language, which means the communities of developers are small, increasing slowly. pyuvm uses Python which itself is an interpreted language, meaning its runtime resource requirements do not scale well on large projects.

2.3. The case against UVM and the mighty ABV

Chip verification is inherently done by hardware designers and engineers and UVM is inherently created by and for hardware designers. When software engineers verify their software, they use unit tests and assertions along with formal proofs. In recent years these approaches have been adopted by hardware designers. *Assertion Based Verification*, or ABV, along with formal verification is increasingly being applied to complex computing chip design verification and increases performance metrics over classic UVM[2]. Key impacts from this paper is summarised on Table 1.

Table 1: *Key impact of formal verification adoption over UVM[2]*

Verification cycles reduced by 25-30%
Pre-silicon bug detection rates improved by 20%
Security vulnerability detection increased by 40%

[1] On GitHub: github.com/chipsalliance/chisel
[2] On GitHub: github.com/SpinalHDL/SpinalHDL
[3] On GitHub: github.com/pyuvm/pyuvm
[4] cocotb on the web: www.cocotb.org/

SystemVerilog Assertions, or SVA has been defined in IEEE 1800-2023[1, Chapter 16] meaning ABV is already a part of the SystemVerilog syntax.

2.4. The goal of this project

The goal of this project is a framework written in a strongly typed language, supporting SystemVerilog and core ideas from ABV, thus making it easy for designers to write their designs in SystemVerilog and use a well known language to implement assertion based tests.
Introducing *SteelBrew*, the chip verification framework written in Java.

3 Problem specification and analysis

As mentioned in Section 2 this project aims to implement a testing framework for SystemVerilog designs, using core ideas of Assertion Based Verification, written in Java. This poses the following challenges:

- **Simulation driver:** The framework needs to communicate with some simulation driver.
- **Peek-poke-step:** Signal manipulation needs to be implemented to set up and carry out assertion of behaviour.
- **Assertion-logic:** ABV logic has to be implemented in the framework.
- **Test translation:** The framework has to translate test logic from Java to SystemVerilog and create a testbench for test simulation.
- **Coroutines/concurrency:** As assertions can be time-invariant, it makes sense to implement concurrency or coroutines to efficiently execute the simulation.

3.1. Simulation driver

Driving the simulations is not part of the scope of this project. Instead, the simulations are run by a third-party tool using Java to invoke the tool and listen for the results. Verilator[5] is a fast and community-driven simulator for SystemVerilog, which supports SVA directives. Through testbenches it is possible to set up and test DUT's fairly easily.

3.2. Peek-poke-step

Using the Java BigInt class, some logic for manipulating bits needs to be implemented in order to set up tests. BigInt enables describing integers in other bases, as is common in hardware design or low-level programming.

3.3. Assertion-logic

There are some assertion directives of interest:
1. Assert which raises an exception if some property does not hold.
2. Cover which monitors the coverage of some property.

These are the minimum that should be implemented. Some object for a test should be created and react if the driver raises an exception pertaining to the used directive. The logic in the framework needs to adhere to the logic stated in the SVA documentation.

3.4. Test translation

Manipulation and assertions has to be properly translated into a testbench that, when run, carries out the verification. This should be done in a manner that translates the test logic into a series of strings that are placed correctly within a testbench.

3.5. Coroutines/concurrency

To optimise runtime, it is desirable to run tests concurrently. For this purpose there are two options:
1. Java Runnable
2. Coroutines

[5] Verilator on the web: veripool.org/verilator

3.5.1. Java Runnable The Java Runnable interface[6] enables thread creation and joining. Benefits include the native support for Java and the documentation and community surrounding developing with Runnable. The main disadvantage is that threads rely on the operating system's scheduler, thus removing some degree of control during execution, and in the worst case, if not run in an elevated mode, threads might not get priority to run.

3.5.2. Coroutines Coroutines are described as "light threads" and enables a thread to control its own behaviour. This means threads can execute with a high degree of control. Disadvantages is that Java does not natively support coroutines. Solutions include hacking the JVM or using third party projects to execute coroutines. One such example is Javaflow[7] from Apache Commons or an interesting project called "Coroutines"[8] inspired by Javaflow.

3.6. Goals and success criteria

The discussed challenges and their solutions are listed on Table 2.

Table 2: *The projects challenges and success criteria*

#	Challenge	Success Criteria
1	Simulation driver	Logic that launches Verilator and handles communication with the processes.
2	Peek-poke-step	Logic that, using BigInt, manipulates wires and ports in the tests.
3	Assertion-logic	Logic that sets up assertion directives supported by the driver and SVA to support ABV.
4	Test-translation	Logic that translates the written test into a testbench that when executed, ensures the tests are carried out correctly.
5	Coroutines/concurrency	Implementation of concurrent execution of the simulations.

SteelBrew will be developed in an Agile way, based on use-cases. This ensures that a viable product is ready after dealing with challenge 1, 2 and 4 as these deals with setting up basic test functionality. Afterwards assertion logic, concluding with coroutines.

4 Design

The design of SteelBrew will take an Agile approach to get going with some framework for writing and running simple tests on devices written in SystemVerilog, using Java.

4.1. Use cases

Using SteelBrew should be as simple as possible. Fig. 1 shows an overview of the use cases for a developer. This is a simplification. As mentioned the framework is executed in Java, so to get started the user would start by creating a java project and importing SteelBrew. Hereafter the DUT is added and tests are defined and configured. A command should then execute the test and start the simulation.

4.1.1. Adding the device under test With the device design in the root folder, adding it to the framework should be as easy as adding the device to some object that handles device configuration. The underlying logic should take care of the rest.

4.1.2. Adding tests Since the tests are linked to some device, it is preferred that the same underlying logic ties tests and devices.

4.1.3. Configure tests This is the bulk of the interaction for the user. With some object carrying the tests and device, the user should be able to easily add new tests or assertions with some easy to use method calls. The underlying logic should resolve conflicts and handle test creation without the user getting much involved.

[6] Documentation on Oracle: docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html
[7] Javaflow on Apache: commons.apache.org/sandbox/commons-javaflow/
[8] Coroutines on GitHub: github.com/offbynull/coroutines

Figure 1: *Use case diagram for SteelBrew*

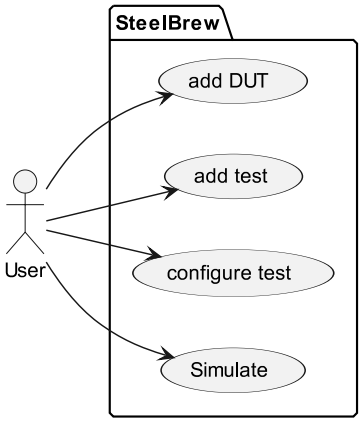
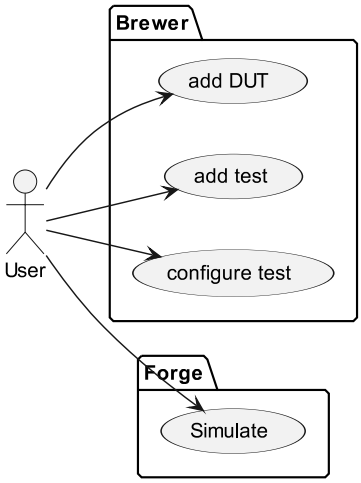


Figure 2: *The expanded use case diagram for SteelBrew*



4.1.4. Simulate Since the simulations are carried out by a third-party program, all handling between SteelBrew and said program, should be handled by yet another class, separate from the test-driven class. This ensures good separation of responsibility in the project.

4.2. Separation of responsibility

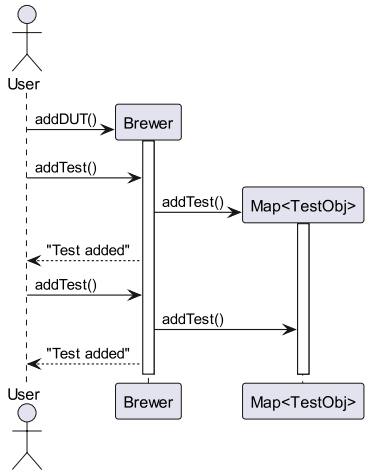
The project is considered two-fold:
1. Handling DUT's and tests
2. Handling concurrency and third-party software

For this reason the program has to contain two main components. The *Brewer* and the *Forge*. A third class is introduced as the entry-point for the program, and will be accessible to all other classes. This class is simply *SteelBrew*. Expanding on Fig. 1, the use case diagram is expanded as seen on Fig. 2.

4.3. The Brewer

The Brewer is the primary interface. There are several ways of going about this object handling multiple DUT's and tests. One aspect could be to hold Lists or Maps of the tests and devices, and then use methods to get these, and couple them accordingly. This have the advantage that from a schematic point of view it simplifies the program. However, once tests are applied to different devices, all contained within the same object, one has to keep track of internal links. The first major decision in the design phase was as such:
1. Give a Brewer to each DUT
2. Assign tests to each Brewer

Figure 3: *Sequence diagram showing the idea behind the Brewer*



This means more objects are created, but the Brewer object itself has a lot less responsibility in terms of tracking the interal links.
A squence diagram on Fig. 3 shows this initial idea.

4.4. The Forge

The Brewers complement, the Forge handles another job, entirely. This class has to handle the responsibility of taking the tests from the Brewers and hand them down to the third-party tool. This prompts some considerations, namely how to launch other programs with very specifik parameters and how to communicate with these programs.

4.4.1. Verilator and the Forge The third-party software chosen is Verilator. It is opensource and very quick at executing simulations. Verilator works like this:

1. The user prepares testbenches describing the testing environment and the tests
2. Verilator takes the testbench and some DUT as input and makes the files for compiling a C++ program
3. Compiling and running the program executes the tests and outputs a waveform-file, which can be inspected by e.g. GTKWave.[9]

This puts constrains on the design of the program. The Forge needs to call Verilator multiple times and then needs to run the compiled program. To ease this, the program creates a Makefile and executes through this. Simple redirects of stdout means that the Forge can collect the output from Verilator and the simulation.

4.4.2. Concurrency Up until now the main idea was to take a Brewer pointing to a DUT, add some tests and then hand it off to the Forge for Verilating, compiling and running the simulation. This poses one issue: This way of evaluating the program means that tests can only be parralised in terms of Verilator's internal multithreading, and then one DUT with all it's test per thread. Verilator's internal multithreading is outside the scope of this project, so the only options is to get creative with the way tests are run. Normally when using Verilator, one testbench with all testing is run per DUT. The design choice here was to split up the tests, such that each test, or set of tests, gets its own testbench and its own compiled program. This way tests can be carried out in parralel by launching each workflow on different threads.

4.4.3. Coroutines It was of great interest to involve coroutines in the project. Java does not support coroutines natively so workarounds were investigated. To this end, it turns out most coroutine frameworks and plugins are 6+ years old. They do not support newer Java versions, and thus not newer Java functionality. The design choice here was to use threads to execute tests.

4.5. Testbench

The workflow of Verilator and the design choice of splitting up tests into separate executables meant that the definition of tests were somewhat straightforward. The Brewer would have Testbench objects. Whenever tests are defined they are added to this object, and just before simulation, testbench files are created based on the content of the object. The testbenches have to have unique names, otherwise the Verilator executable and GNU Make will be unable to execute the correct benches.

[9] GTKWave on the web: gtkwave.sourceforge.net

4.6. Batch

While the Testbench class is able to contain all tests put in, it is basically a collection of strings. Furthermore, it is often desireable to control the sequence in which tests are executed, as the result from one test might affect the other. Or if one has to test what happens when manipulating signals inside the DUT. A new class is introduced to handle a sequence of tests, along with logic transforming simple methods into cumbersome strings for the testbench. The Batch class keeps a record of signals and tests involved in some scenario and when ready, creates a testbench with these tests executed as they are added by the user.

4.7. Assertions

5 Implementation

Document implementation

6 Testing

Insert and describe tests on framework

7 Conclusion

Write up conclusion

References

[1] "IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language". In: *IEEE Std 1800-2023 (Revision of IEEE Std 1800-2017)* (Feb. 2024), pp. 1–1354. DOI: 10.1109/IEEESTD.2024.10458102. URL: https://ieeexplore.ieee.org/document/10458102.

[2] Kaushik Velapa Reddy. "Formal Verification with ABV : A Superior Alternative to UVM for Complex Computing Chips". In: *International Journal of Scientific Research in Computer Science, Engineering and Information Technology* 10.6 (Nov. 5, 2024). Number: 6, pp. 90–98. ISSN: 2456-3307. DOI: 10.32628/CSEIT24106157. URL: https://ijsrcseit.com/index.php/home/article/view/CSEIT24106157.

List of Figures

List of Tables

Listings