Designing a VHDL Toolchain

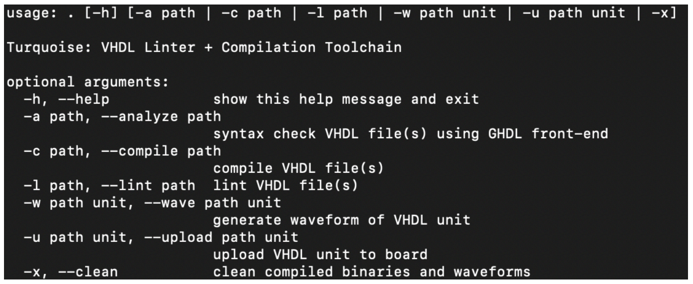
By Athokshay Ashok, ECE '21

Introduction

Like most popular programming languages used today, VHDL (Very High-Speed Integrated Circuit Hardware Description Language) has a familiar syntax that enables programmers to describe digital circuits through code. For someone new to the world of HDLs, it is common to make syntactic and semantic mistakes that can often result in vague error messages from the standard VHDL compilers used in widely used products such as ModelSim and Radiant. Linting is a static code analysis process that runs separate from the compilation stage and attempts to catch these mistakes in order to provide more constructive warnings and errors. Our linter focuses primarily on checking for consistency with types for declared signals, initialization of declared signals, entity-component mismatches, and undeclared libraries.

Related Work

There are several commercial VHDL linters available in the market such as VHDL-Lint and Sigasi, but they lack the simplicity that our project offers in terms of usage and specificity of targeted error messages. Additionally, we provide a cross-platform toolchain for generating waveforms of testbenches and uploading programs to an FPGA. Any function can be performed with a single command from the CLI, while existing tools often require multiple steps to achieve the same tasks.



```
usage: . [-h] [-a path | -c path | -l path | -w path unit | -u path unit | -x]

Turquoise: VHDL Linter + Compilation Toolchain

optional arguments:
  -h, --help            show this help message and exit
  -a path, --analyze path
                        syntax check VHDL file(s) using GHDL front-end
  -c path, --compile path
                        compile VHDL file(s)
  -l path, --lint path  lint VHDL file(s)
  -w path unit, --wave path unit
                        generate waveform of VHDL unit
  -u path unit, --upload path unit
                        upload VHDL unit to board
  -x, --clean           clean compiled binaries and waveforms
```

Figure 1: CLI commands for our toolchain

Toolchain

Our toolchain, which is built entirely on Python, uses the following open-source tools: 'GHDL' for VHDL compilation and elaboration, 'gtkwave' for reading in .vcd files and generating waveforms, 'yosys' for synthesis, 'nextpnr-ice40' for routing, 'icepack' for generating bitstreams, and 'iceprog' for uploading bitstreams to an FPGA. We use subprocess calls to run the executables for each tool in sequence seamlessly, with the only required inputs being a file path and/or a unit name. If any step in the pipeline fails, we terminate and return constructive error messages as to which step failed and the most likely reason for the failure. These executables run much faster than the compilers and waveform generators provided by ModelSim and Radiant. Currently, we only support uploading bitstreams to an UPduino 3.0 board, which is the primary board used by the ES-4 course at Tufts.

Tokenizing a VHDL File

To parse through a VHDL file, we need to convert the raw text into meaningful information. This is done using 'pyVHDLParser', an open-source python library that slices a VHDL file into tokens and returns an iterator for the stream of generated tokens. The tokens indicate whether something is a space, a line break, a comment, or a keyword that indicates the start of a critical section of the file. Each token also contains information such as the file name, line number, and column number where the token starts, which enable us to output clear messages. The iterator itself is a double-linked list that starts at the head of the file.

Using a DFA

Every VHDL file has to follow a specified format: a list of imported libraries, an entity declaration, and the architecture of the unit. Within the entity, we declare signals and their types in ports and as generics. In the architecture, we can include components implemented in other files, declare new signals, create processes, and perform other computations. A single missing bracket, comma, colon, or semicolon can cause the compilation of a file to fail, so it is crucial to track the occurrences of the tokens and store this information for future checks.

To keep track of what state a file is in as we iterate through the tokens and to check if there are any potential errors, we use a deterministic finite automaton (DFA), which is a state machine. Every DFA must have exactly one start state, from which we can add transitions to other states depending on which inputs we receive. There exists a set of accept states that indicates a successful traversal through the machine, given that there are no more tokens expected.

We first designed a class that represents a single state, and then an interface for a DFA that can be used to create state machines for each specific section of a file by simply adding new states and transitions. We also include callback functions, which are triggered if the machine makes a certain transition. This enables us to compare values from the current state to those derived from the previous states.

Constructing Customized DFAs

With this template, we created DFAs for each sub-component of a VHDL file such as entity, component, signal, port map, etc. Figure 2 shows the DFA for any entity declaration.

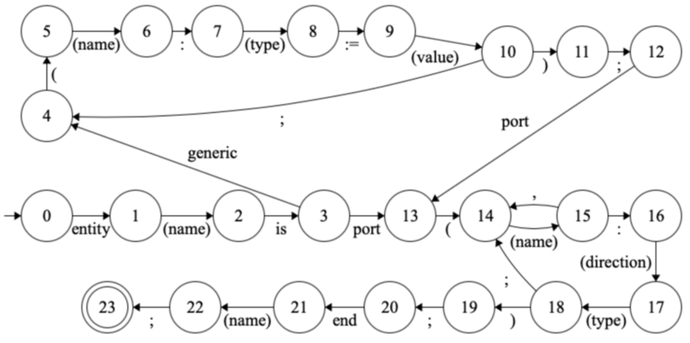


Figure 2: DFA for Entity declarations

To track errors and warnings along the way, we created classes to represent them as well as a logger class to store each entry. A messages class was also implemented to print color-coded messages to the CLI using the ‘colored’ library from Python.

The DFAs each take in as input the token iterator and the logger. At the top level, the DFAs are triggered by certain keywords. For example, if we encounter the keyword “entity” as we iterate through the tokens, we call the parsing function from the entity DFA and have it run until it reaches an accept state or encounters an error/warning.

We can also call an another DFA inside the current one. For instance, the entity, component, and signal declaration DFAs all require a type-check of the signals. We modularized this process by creating a DFA that gets triggered when it reads a signal type token and verifies if it has the right format. We support most of the commonly used types like std_logic, std_logic_vector, unsigned, signed, etc. After this state machine has processed the type and reached an accept state, we return to where we left off in the main DFA and resume traversing.

We run through the files twice: once to store the necessary information and once to validate information between multiple files.

Conclusion

VHDL can be tricky to get accustomed to, especially for those who have prior experience with some standard programming language such as Java, Python, C++, etc. While there are plenty of software that offer compilation and simulation, our project aims to simplify these processes, provide better insights into errors, and support both MacOS and

Windows. Our linter can be easily scaled up to perform further static code analysis and target a range of different errors.

References

1. Gingold, T. (2020). *GHDL Documentation Release 1.0-dev*. https://ghdl.readthedocs.io/_/downloads/en/latest/pdf/

2. *GTKWave 3.3 Wave Analyzer User's Guide*. (n.d.). Retrieved May 6, 2021, from http://gtkwave.sourceforge.net/gtkwave.pdf

3. Lehmann, P. (2020, December 27). *The pyVHDLParser Documentation — pyVHDLParser 0.6.0 documentation*. Pyvhdlparser.readthedocs.io. https://pyvhdlparser.readthedocs.io/en/latest/

4. Wolf, C. (n.d.). *Yosys Manual*. Retrieved May 6, 2021, from http://www.clifford.at/yosys/files/yosys_manual.pdf

5. Sipser, M. (2012). Introduction to the Theory of Computation. United States: Cengage Learning.