Turquoise Blue: Linting Toolchain for VHDL Programming

Linting And the Roles Played by the Static Code Analysis Methods and Tools in its implementation.

By Siegfred Madeghe, ECE '21

Introduction

Linting is an automated checking of a source code for programmatic and stylistic errors. In computerized systems, the linting implementation uses a tool known as a linter, which analyzes computer code without executing it.

Historically, linting started in 1978 after being invented by a computer scientist—Stephen C. Johnson—while working at Bells Labs. Stephen was debugging the Yet Another Compiler-Compiler (YACC) grammar program that he was writing for the C programming language when he invented linting. Since then, the invention of this technology has been used widely across different programming languages to analyze and generate computer programs quickly.

The development of this tool has now enabled programmers—whether working for software or hardware systems (using Hardware Description Languages)—to "concentrate at one stage of the programming process solely on the algorithms, data structures, and correctness of the program, and then later retrofit, with the aid of lint, the desirable properties of universality and portability" [2].

Hardware Description Language

Hardware Description Languages (HDL) are specialized computer languages used to describe the structure and behavior of electronic circuits. HDLs have improved the speed and efficiency in designing and building digital systems; thus, accelerating the development of complex digital systems.

There are two main hardware description languages: Verilog and Very High-Speed Integrated Circuits (VHSIC) Hardware Description Language, shortened to VHDL. Both Verilog and VHDL implement register-transfer-level (RTL) abstractions; however, VHDL is more verbose than Verilog. Therefore, VHDL designs are easy to understand and often catch errors missed by Verilog.

It is crucial to understand that HDL codes, like other programming codes, are prone to errors. This observation is factual, although HDLs programming flow is unlike the usual software programming languages. For computer codes generated by general programming languages execute serially while those by HDL execute in parallel. Given the similarity of the problem—the need to produce error-free code—it is then possible and needful to implement linting tools for HDL.

Factors to Consider Before Implementing a Linter for HDLs

A linter for an HDL platform must consider these crucial criteria: First, designers must decide what kind of errors the linter should detect. Second, the type of static code analyzer to link with the linter. Third, error reporting methodology. And, fourthly, the protocols used to upload the reviewed code into the specified electronic device.

Among these four criteria, however, the first two are the most important ones—because specifying the types of errors for the linter to analyze helps to scope the tool's functionalities. As a result, determining the supporting tools to be incorporated into the linter becomes easier: more errors need a more powerful static code analyzer.

Static Code Analyzer

A static code analyzer is a tool that aims at exposing possible vulnerabilities in a computer program before its execution. Technically, it is a good design strategy to implement the static code analyzer before the compilation process. Because "roughly half of the security [and programming] weaknesses get introduced during coding" [10]. A source code must find weaknesses and prepare a report used by other tools to report and fix the found errors. Depending on the implementation of the static code analyzer, the development of a mechanism to suppress false error flags to enhance practicability in repeated code scanning is paramount [1][6][8].

It is crucial to note that "False positives are a critical factor in static code analysis" [10]. Because static code analysis issues are naturally 'undecidable': computed model approximations lead to misses, arising when the static analyzer misses the source code weaknesses or report the correct code as a weakness. Therefore, in choosing a static analyzer tool, designers must choose tools with acceptably low false-positive rates.

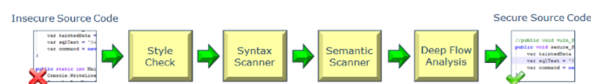

Figure 1: Code Analyzer Flow [3]

Types of Static Code Analyzers

The implementation of a static code analyzer can take different forms. But majorly, static code analyzers exist in two classes: analyzers that directly analyze source code and analyzers that analyze compiled byte code. Direct code analyzers work 'directly' on the source code written by the programmer. This kind of implementation is always beneficial for source codes with less than a hundred thousand lines of code. Therefore, this choice makes it better for the static code analyzer for small projects demanding at most a thousand lines of code.

On the other hand, the compiled byte code is relatively faster because it works on a compiled code. This kind of implementation is better for programs with more than a thousand lines of code. Regardless of the implementation method, though, both options share the same logical flow: They both inspect the program and formulate an abstract representation of the program used to match the selected error scheme. Moreover, in a static code analyzer, equipping the tool with data-flow analysis capabilities improves the analyzer's ability to perform vulnerability checking. This addition makes the tool robust towards any security vulnerabilities to be exploited by hackers.

Static Code Analyzer to Consider in the Making of a Linter

A linter can use static code analyzers like Flex and Bison in its implementation. Flex and Bison are tools that build programs that handle structured inputs. Initially, they were the fundamental building tools for compilers. However, recently, they have been used as building blocks in developing tools like parsers and static code analyzers. A static code analyzer scans the parsed code is for errors. This scanning process works "by looking for patterns of characters in the input" [9]. A direct way to explain the theory behind these patterns is by using "regular expression, shortened to reflex or regexp" [9].

Flex programs include a list of regexps with instructions that tell the program what to do when the input code matches them. This list is called actions. Therefore, in summary, a flex scanner inspects the input code by comparing it with the regexps included in the list—actions—and then giving an appropriate message in case of mismatches.

Moreover, flex scanners produce a list of tokens handled by a parser. Hence, every time a program inquires for a token, a function call is called, and then tokens are returned. Each of these tokens has two parts: the token itself and its value. There is no specific structure to how the token values are assigned, "except that the token zero means end-of-file" [9]. So, when Bison generates a parser, it generates token numbers as well, beginning from 258. This token generation process happens automatically, and it is effective for "it avoids collisions with literal character tokens" [9][5].

Constructing a Good Linting Tool

The qualities of a robust linting tool are diverse. But, in general, a well-designed linting tool is safe, easy-to-use, reliable, and effective. These characteristics must be thought of and systematically incorporated into the tool's design from the beginning of the designing process [4].

However, depending on the quantity and complexity of the errors that the linter will lint, a linter can make use of open toolchains in its implementation. These toolchains provide already linked tools that aid in reducing production time while increasing security compactness. The linter should also incorporate UI/UX tools that improve usability—since a smooth user experience should also be a goal that the linter must target.

Also, given that the static code analysis is rule-driven, it is crucial to ensure proper implementation of the design rules. This demand provides assurance and safety compliance protocols in designing digital systems; therefore, improving code security—which is a critical issue recently, as it is pivotal to analyze code for potential vulnerabilities from different perspectives [11][6][7].

Condusion

Designing digital systems using computers is challenging, especially now that there has been a profound rise in security vulnerabilities. Also, for beginners, hands-on experience in software that implements HDL is critical. Therefore, this software must provide thoughtful, easy-to-understand error messages caught by the static code analyzer.

Reference

1. Chelf, Ben, and Christof Ebert. "Ensuring the Integrity of Embedded Software with Static Code Analysis.", May/June 2009 7. Black, Paul E. "SAMATE and Evaluating Static Analysis Tools.", September 2007.

2. https://en.wikipedia.org/wiki/Lint_(software)

3. https://www.softscheck.com/en/security-consultancy/static-source-code-analysis/

4. Chinchani, Ramkumar, and Eric Van Den Berg. "A Fast Static Analysis Approach to Detect Exploit Code Inside Network Flows.", 2005.

5. Bush, William R., Jonathan D. Pincus, and David J. Sielaff. "A Static Analyzer for Finding Dynamic Programming Errors.", December 1999 11. Bergeron, J., M. Debbabi, J. Desharnais, M.M Erhioui, Y. Lavoie, and N. Tawbi. "Static Detection of Malicious Code in Executable Programs.", 2001

6. Holzmann, Gerard J. "Conquering Complexity." Computer 40 (12): 111-113, Dec. 2007. 13. "Source Code Security Analysis Tool Functional Specification Version 1.0." National Institute of Standards and Technology (NIST), Special Publication 500-268. May 2007.

7. https://techbeacon.com/app-dev-testing/3-steps-aligning-client-side-static-code-analysis

8. https://softwareengineering.stackexchange.com/questions/27682/what-are-the-real-benefits-of-static-code-analysis

9. https://www.oreilly.com/library/view/flex-bison/9780596805418/ch01.html

10. https://www.veracode.com/security/static-code-analysis

11. https://www.perforce.com/blog/qac/what-lint-code-and-why-linting-important

12. https://barrgroup.com/Embedded-Systems/How-To/Lint-Static-Analysis-Tool