SystemVerilog Productivity Tools

David Fang
fangism@google.com

***Abstract*—Google is a software company with outstanding productivity support (searching, indexing, building, analyzing, formatting, editing) for its core languages: C++, Python, Go, Java, and JavaScript. Internal productivity research has measured the impact of productivity-enhancing tools on software engineers to be positive and significant, enough to justify staffing the development and maintenance of these tools and services. Notably absent from the list of supported languages is SystemVerilog, which is important to the growing hardware design and verification community at Google. We present some tools that have been developed to aid productivity working with SystemVerilog: a style linter and code formatter. Both of these tools aim to be extensible and customizable, and would be valuable to organizations keen on enforcing their own style guidelines. These tools depend on only open-source tools and libraries, such as Abseil-C++ [1], and are planned for release to the open-source community in the future.**

I. Style Guide

At Google, hardware engineers have collectively written a coding style guide for SystemVerilog. The style guide describes formatting how to format consistently (indentation, spacing, wrapping), and best practices about using certain SystemVerilog constructs. Externally, to our collaborators, it sets a standard to follow for sharing source code and IP. The SystemVerilog style guide is actively being prepared for initial public release. This follows Google's precedent for publishing style guides for other languages [2].

Examples from Google's SystemVerilog Style Guide:

Wrap lines at 100 characters.
Tabs are forbidden, use spaces.
Indent 2 spaces per indentation level.
Delete trailing whitespace at the end of lines.
Do not use `` `uvm_warning ``*. Prefer to mark something as an error or fatal.*

Without a style checking tool, identifying style violations during code review remains a cognitive burden for both authors and reviewers. Without a formatter or a decent editor, engineers waste time pushing spaces around. It is also not productive to debate whose editor formats better. Our new tools fulfill the need for productivity aids.

II. Tool Requirements and Considerations

Certain code services (internal to Google) impose the following requirements on code that runs in production environment:

- Source code is available for audit.
- Built binaries can run without restriction on production servers (e.g. license-free, or with site-license).

Linter requirements:

- Tool must operate on one file at a time in isolation.
- Tool must operate on un-preprocessed source code.

Formatter requirements (in addition to linter requirements):

- Must preserve all original text including comments, possibly whitespace.

Most compilers for preprocessed languages like C are designed to preprocess in a separate phase before lexing by expanding file-inclusions, selecting conditional text, and expanding macro definitions, however that phase separation is not as well-suited for the single-file linter and formatter, so adapting existing SystemVerilog parsers did not seem like a good fit. We were unable to find a pre-existing solution that met all requirements.

When considering building our own front-end, the greatest unknown was the cost of developing and maintaining our own parser for a language as complex as SystemVerilog. If successful, owning our own toolchain would empower us to enforce our own style guide without depending on a third party for support. Properly designed, our tools will be extensible and configurable, making them usable to hardware communities outside of Google. We hope that by open-sourcing these tools, we will raise their quality by testing against a wider variety of source code, and growing a community of users and contributors around them.

III. Lexing and Parsing Library

The common feature that ties all of these tools together is the front-end lexer and parser. Parsing SystemVerilog is non-trivial due to the size and complexity of the language; the appendix of the SystemVerilog Language Reference Manual that summarizes the keywords and legal syntax is over 40 pages long [3]. This fact alone is enough to deter most efforts to create a parser, let alone spend the engineering effort to create a freely available one.

The initial version of the lexer and parser was based on those from the Icarus Verilog simulator [4], which use the Flex lexer generator [5] and GNU Bison, an LALR(1) parser generator [6]. Over time, the grammar has been extended to support most of SystemVerilog – it now accepts over 98% of the corpus of SystemVerilog files from real projects at Google. To achieve this, we made grammar approximations and simplifications, that may result in accepting some invalid syntax. Accepting valid syntax is far more important than rejecting invalid syntax because this toolchain is not on the production path to create or verify hardware designs. To regain the precision lost by syntactic approximation, one could check

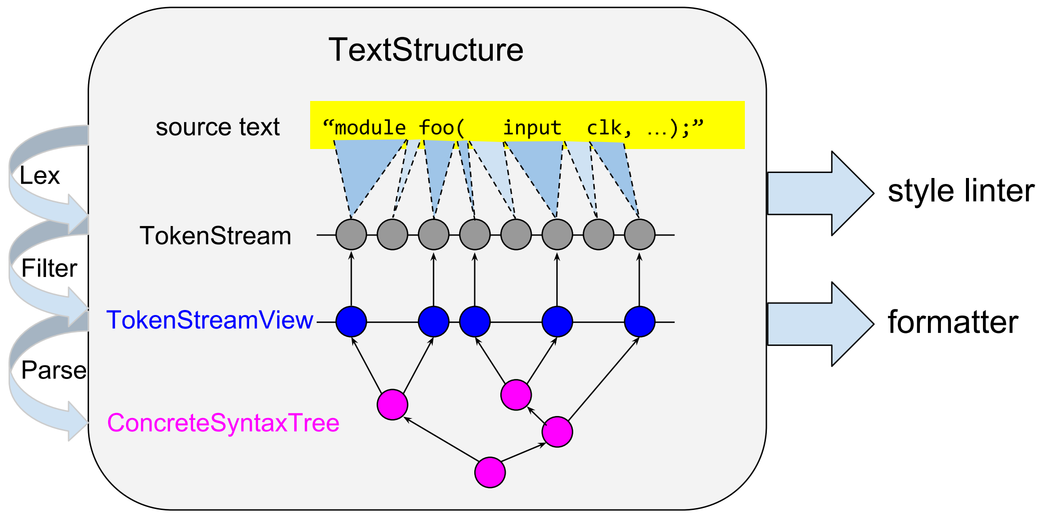

Fig. 1. A decoupled lexer and parser architecture allows for greater flexibility in creating different filtered views on a raw token stream. For example a parser may choose to ignore comments, but a formatter must preserve comments.

additional constraints in the semantic actions during syntax tree construction, or check the resulting syntax tree in a separate pass. However, this was deemed low priority because other production tools (although more costly to run) will catch the rare invalid code that this parser allows. Every improvement to the parser included the appropriate SystemVerilog input test cases to ensure that quality would never regress.

The earliest manifestation of the SystemVerilog parser was an outline-generator for web-based source code navigation, which is run by a service that periodically crawls the code base. In the current implementation, preprocessing directives are integrated directly into the lexer and parser (unlike production compiler toolchains), however, this means that constructs such as `` `ifdefs`` are only accepted in limited locations. Other preprocessing strategies (such as assume-undefined) are possible, but unimplemented. All macros calls (e.g. `` `uvm_error(...)``) remain un-expanded, and have their own rules in the parser's grammar, and nodes in the resulting syntax tree. Macro calls are supported in limited, but common positions, such as expressions, body items, and statements.

The SystemVerilog lexer and parser evolved into a standalone C++ library that produces a sequence of tokens, and a concrete syntax tree in language-agnostic data structures (akin to LISP structures) with language-specific enumerations. The raw token stream includes white-spaces, comments, and attributes, however, parsing and tree construction operate on a filtered subset of tokens. All syntactic constructs are represented as generic tree nodes with type enumerations. For example, the code in Listing 1 yields an internal syntax tree representation described in Listing 2. A nonterminal node in the grammar such as `module_declaration` is designated with a tag `kModuleDeclaration`. Leaf nodes correspond to tokens, so in the example listings, `input` is a keyword token that appears at byte-offset 17–22, and `clk` is an identifier token that appears at byte-offset 23–26.

Listing 1. Example code

```
module flop(
    input clk,
    input d,
    output reg q
);
  always @(posedge clk)
    q <= d;
endmodule : flop
```

Listing 2. Concrete syntax tree representation of Listing 1, truncated.

```
Node (tag: kDescriptionList) {
  Node (tag: kModuleDeclaration) {
    ("module" @0-6: "module")
    (Identifier @7-11: "flop")
    Node (tag: kModuleHeader) {
      ('(' @11-12: "(")
      Node (tag: kPortDeclarationList) {
        Node (tag: kPortDeclaration) {
          ("input" @17-22: "input")
          (Identifier @23-26: "clk")
        }
...
```

Figure 1 illustrates the high-level design of the front-end parsing library. Lexing is done separately and produces a stream of tokens that includes whitespaces and comments. A separate filtering phase presents a view of tokens that the parser then arranges into a concrete syntax tree. The separation between the lexer and parser facilitates the construction of different structured views of the source code. The phase between the lexer and parser could be used for preprocessor expansion, or annotating the token stream with contextual information to aid in parsing.

Currently, the parser library lacks a SystemVerilog-specific abstract syntax tree (AST) classes and symbol tables, i.e. there are no classes for language-specific constructs like `module`, `function`, `task`, and `class`. Without language-specific classes, there is no semantic analysis or elaboration. The parser library currently lacks a real preprocessor. The features it has today are sufficient to build the style linter and formatter, which operate entirely on the language-agnostic structural representations of the code.

IV. Style Linter

The style linter analyzes the code structure (e.g. line-by-line, syntax tree) to find patterns that have been defined as style violations. Syntax trees are analyzed with a tree pattern matching library that allows for compact description of patterns of interest, and is inspired by Clang's AST Matcher library [7]. For example, a C++ expression resembling `Node_GateModuleInstance(HasPortList())` would be sufficient to produce a search that matches module instances with port connections. Further search refinements and analyses can be performed on matched subtrees in C++ code. Violations are collected and reported to the user.

Presently, only around 10 SystemVerilog style violation rules have been implemented, although over 50 rules have been proposed by the community, many of which cite our own style guide. Each rule is implemented in its own C++ file, designated with a name, and registered with a global registry that controls which rules are enabled.

The style linter also features a comment-based waiver mechanism for waiving violations on individual lines or ranges of lines. Style lint waivers can apply to the next non-comment line (which makes them stackable), or apply to a range of lines.

```
// verilog_lint: waive rule-name
... waives violations on this line only

// verilog_lint: waive rule1-name
// verilog_lint: waive rule2-name
... waives two types of violations here

// verilog_lint: waive-start rule-name
... waives one violation type
... in this range of lines
// verilog_lint: waive-stop rule-name
```

Since the syntax tree data structure is language-agnostic, the analyzer framework and matcher library supports creating style linters for other languages. The SystemVerilog style linter plugs into some of Google's internal source code services that automatically run on code under review or in web-based editors [8]. We have foregone analyzing most formatting violations because those are expected to be reported and fixed by the formatter tool.

V. Formatter

The role of the formatter tool is to rewrite code to conform to a defined style – placing spaces in a globally consistent manner, which aids in readability and reduces the friction in code reuse across projects. The tool formats syntactically valid code, and leaves invalid code unformatted. Currently, the tool does not attempt to recover from syntax errors, because the error recovery path leaves gaps in the resulting syntax tree.

The architecture of our formatter is inspired by `clang-format` [9]: the token sequence is partitioned into slices (unwrapped lines) that are processed independently. Each slice is optimized over a set of line-break and spacing decisions that minimize a cumulative penalty metric. However, whereas `clang-format` only requires lexical information, we chose to analyze the structure starting with a fully-formed concrete syntax tree. The major advantage here is not having to create or maintain a separate parser from one used by the compiler. As we traverse the syntax tree, we can know the full context under which every token appears, and make informed decisions accordingly.

We considered trying to leverage `clang-format` itself, but mapping SystemVerilog constructs onto the LibFormat (intended to support multiple languages) seemed cumbersome, and involved maintaining a separately written parser [10], [11]. While some syntax in SystemVerilog is C-like, there are enough grammar differences to deter us from duplicating parser effort. At the beginning of the formatter project, we already had a mature parser and syntax tree, so we decided to leverage that instead.

The formatter tool is still under development, and yet to undergo wide internal testing. Once completed, it is expected to be invokable from any modern editor, and plug into existing source code services.

VI. Roadmap

Google's SystemVerilog style guide is actively being prepared for external initial release. Development plans for the SystemVerilog parsing library include various bug fixes, and improved preprocessor support. The style linter is missing a lot of useful rules requested by our own engineers. The formatter is in its infancy and will need to undergo a lot of internal testing before an initial release. Over time, we plan to make the formatter highly configurable. Releasing the style linter and formatter tools to the open-source community still requires internal approvals, and funding for staffing.

VII. Acknowledgements

Jeremy Colebrook-Soucie implemented the style linter as a summer intern in 2017. Cameron Korzecke implemented the formatter as a summer intern in 2018. Sergey Sokolov co-hosted and mentored both interns. Thanks to Icarus Verilog for providing a starting point for the lexer and parser.

References

[1] "Abseil common library." http://github.com/abseil/abseil-cpp.

[2] "Google style guides." http://github.com/google/styleguide.

[3] "IEEE 1800-2017 - IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language,"

[4] "Icarus verilog simulator." http://icarus.verilog.com/.

[5] "Flex lexer generator." http://github.com/westes/flex.

[6] "Bison parser generator." http://www.gnu.org/software/bison/.

[7] http://clang.llvm.org/docs/LibASTMatchersReference.html.

[8] C. Sadowski, J. van Gogh, C. Jaspan, E. Söderberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, (Piscataway, NJ, USA), pp. 598–608, IEEE Press, 2015.

[9] "Clang format." http://clang.llvm.org/docs/ClangFormat.html.

[10] "Clang format library." http://github.com/llvm-mirror/clang/tree/master/lib/Format.

[11] "Clang format interface." http://github.com/llvm-mirror/clang/blob/master/include/clang/Format/Format.h.