Automated Feature Testing of Verilog Parsers using Fuzzing (Registered Report)

Quentin Corradi
Imperial College London
London, United Kingdom
q.corradi22@imperial.ac.uk

John Wickerson
Imperial College London
London, United Kingdom
j.wickerson@imperial.ac.uk

George A. Constantinides
Imperial College London
London, United Kingdom
g.constantinides@imperial.ac.uk

Abstract

In this article we propose a methodology based on fuzzing to test which features are supported by pasers and register an experiment applying this methodology to SystemVerilog-consuming tools. SystemVerilog is a hardware description, specification and verification language widely used in hardware design, and with an active standard committee. Most SystemVerilog-consuming tools have incomplete support and support additional features. These tools do not provide the list of features they support, so identifying commonly supported SystemVerilog features is complicated. This hinders design portability and tool interoperability. We think current efforts to test these tools' feature support are insufficient. All of the previous points justify why SystemVerilog-consuming tools are a good candidate for our methodology. We also provide the first (to our knowledge) open-source parser and fuzzer for Verilog with full support and compliance with the 2005 standard.

CCS Concepts

• **Software and its engineering → Software testing and debugging**; *Parsers*; • **Hardware → Hardware description languages and compilation**.

Keywords

Fuzz testing, Grammar-based fuzzing, Mutation testing, Verilog

ACM Reference Format:
Quentin Corradi, John Wickerson, and George A. Constantinides. 2024. Automated Feature Testing of Verilog Parsers using Fuzzing (Registered Report). In *Proceedings of the 3rd ACM International Fuzzing Workshop (FUZZING '24), September 16, 2024, Vienna, Austria.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3678722.3685536

1 Introduction

In this article we propose a methodology based on fuzzing to test which features are supported by parsers. We also propose an experiment applying this methodology to test language support of SystemVerilog-consuming tools. We think these tools are particularly suited to test this methodology because SystemVerilog is still a relevant language used in a fragmented ecosystem of tools.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
FUZZING '24, September 16, 2024, Vienna, Austria
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1112-1/24/09
https://doi.org/10.1145/3678722.3685536

SystemVerilog is a widely used hardware description, specification and verification language [4] with an active standardisation committee. Most ASIC and FPGA designs on the market use some Verilog or SystemVerilog [13]. Except in the previous sentence, in this article SystemVerilog will refer to the language as a whole whereas Verilog will specifically refer to the IEEE 1364-2005 [3] standard of Verilog. The recent SystemVerilog standards are sizeable, and most SystemVerilog-consuming tools either do not support it entirely or support additional non-standard features. This is not helped by the lack of available open-source SystemVerilog designs reflecting industry practices. On top of that, most tools do not provide the list of language features they support. Therefore, identifying commonly supported language features is complicated. This is a long-standing issue for design portability and tool interoperability, forcing some users to stick to outdated standards, as underlined last year by the technical chair of the IEEE SystemVerilog Working Group committee:

> "Many users avoid adopting SystemVerilog because feature support from different tools and vendors of the rapidly changing LRM[1]] had been so inconsistent. To this day, people continue using Verilog-1995 syntax and avoid using features added by Verilog-2001 (e.g., ANSI-style ports and the power operator)." – *Dave Rich, Technical chair of the IEEE SystemVerilog Working Group committee* [29]

Knowing the state of support for SystemVerilog in widely used commercial tools would make it possible to know a safe common subset of supported language features. This safe common subset of the language would have several benefits:

- It would make it easier for people to write hardware designs in a subset of the language supported by most tools, and less restrictive than the original IEEE 1364-1995 standard [2, 29].
- It could help to direct development effort of open-source tools to reach feature parity with commercial tools.
- It would help choosing a feature set for bug-finding campaigns [19, 36] and formalisation efforts [24].
- Awareness of the state of support of the language could help tool developers make their tool be inter-operable with other SystemVerilog-consuming tools.

Some efforts have already been made to document feature support of SystemVerilog-consuming tools. The only project we know of is Sv-Tests [6] by CHIPS Alliance. It benchmarks many open-source tools according to which features of the language they support. It is useful to track lacking features of existing open-source parsers and to identify the most compliant available open-source

[1] Language Reference Manual, called "standard" in this article.

parsers so that new tools can choose an existing parser to extend or use as a reference to build their own. However the lack of any systematic testing means they missed some bugs we were able to quickly identify in a pilot experiment, even for the most compliant parser they feature. This also includes non corner-case bugs in categories for which SV-TESTS reports full support, such as the following bug we found in the escaped identifier support of Surelog [12].

Example. The following testcase:

```
module \monkaS ;
endmodule
```

should be accepted, but is rejected because of the capital 'S' at the end of "\monkaS". The cause of this bug is in the grammar used by Surelog's parser:

```
ESCAPED_IDENTIFIER: '\\' [WS\r\t\n]*? WS;
WS: [ ]+;
```

The problem is that instead of excluding white spaces from the character-set after the backslash, the grammar excludes the characters 'W' and 'S'.

The lack of compliant parsers and the fact that the standard's BNF is difficult to interpret means that extending or using as reference another parser will lead to more parsers with compliance issues. Therefore a fully compliant and open-source reference parser like the one we implemented to run our pilot experiment could be helpful to implement correct tools. This is especially true for tools like linters, code transformation tools, and fuzzing tools, which work with arbitrary code and require full language support.

In this study, we plan to partially automate feature testing of many commercial and open-source SystemVerilog-consuming tools using fuzzing. To that end, in Section 2 we explain how we plan to use several fuzzing techniques on these tools' parsers to test the parsers' accepted language constructs. Then in Section 3 we describe a pilot experiment which helped design and justify the feasibility of the experiment we plan to run. Finally in Section 5 we discuss about limitations of our experiments, potential solutions and future directions we will not be exploring.

In summary, the contributions of this paper are:

(1) A way of using fuzzing to partially automate testing languages features supported by a parser for which expected support differs significantly from a reference.
(2) A planned experiment in which we will apply this methodology on SystemVerilog-consuming tools.
(3) The first (to our knowledge) implementation of a fully compliant open-source reference Verilog parser[2].

2 Planned Experiment

2.1 Research Questions

For a feature to be supported it needs to be parsed, its semantics interpreted correctly, and any processing done with it to not be buggy. The first step is correct parsing and this can be tested with fuzzing. In particular grammar based fuzzing seems fit for this purpose. Testing any subsequent part in the support of a feature needs a purpose built tool which requires more effort. To avoid wasting effort, testing that the feature is parsed is an obvious first step. Therefore our first idea to test the SystemVerilog features of a tool is to use fuzzing in a context-free grammar-based-like way. We are not sure context-free grammar fuzzing will be sufficient because the SystemVerilog language is not context-free.

RQ1: Is context-free grammar fuzzing suitable to test supported features?

To answer this question we will run grammar based fuzzers to provide inputs to several SystemVerilog-consuming tools. If the fuzzing campaign is unable to find more bugs because the parsing steps completes without any error related to parsing and the parsing covered the whole input then context-free grammar fuzzing is suitable. We consider the parsing covered the whole input if there are signs of unsuccessful processing other than parsing (like typing errors) on the whole input, or without that, if mutating any part of the input to make it invalid triggers a parsing error. On the other hand if we reach a point in the fuzzing campaign at which, to keep getting parser errors we need to add context awareness to the fuzzing tool, then context-free grammar fuzzing is unsuitable.

Another promising method which may outperform fuzzing for this task is systematic testing. The main advantage of systematic testing is its ability to achieve high coverage of a grammar. The choice of the coverage metrics is important as inputs are minimised when no increase in coverage is expected.

RQ2: Is systematic testing suitable to test supported features?

To answer this question we will perform the same test as grammar-based fuzzing using TRIBBLE [18], a grammar-based systematic testing software using k-path coverage as its coverage metrics. We will compare this solution to fuzzing in terms of bugs found.

Out of this fuzzing campaign we should be able to get a list of unsupported inputs for each tool. We are interested in a subset of SystemVerilog supported by all the tested tools. Deriving this subset should be a matter of reducing the language grammar to not describe any unsupported input:

RQ3: Is a list of non-compliances to the standard enough to derive a useful subset of the language supported by all tools?

The empty input is valid SystemVerilog so there is always a common subset of the language supported by all tools. We are more interested in knowing what can be expressed by this subset. This subset should provide the ability to express sequential and combinational designs. In particular if the subset can be used to encode and synthesise any finite-state machine then we will consider the subset useful.

All of the grammar-based fuzzing methods described before should only find valid inputs rejected by the tools' parsers. To find invalid but accepted inputs we need the ability to generate slightly invalid inputs. This is a generally a hard task because for an invalid input to be accepted, it needs to produce a valid sequence of tokens, and changing some characters in this sequence would make the input rejected. If the number of these characters is n then the probability of finding such an input by random exploration is usually on the order of $\#\{characters\}^{-n}$. To avoid relying on this probability, current approaches generally rely on mutating inputs with knowledge such as coverage, input grammar or valid tokens.

[2] https://github.com/ymherklotz/verismith

One of these approaches is a recent work by Bendrissou et al. [20]. They try with their tool called GMUTATOR to generate slightly invalid inputs by mutating a reference grammar then using the mutated grammar to generate inputs. These inputs are then also mutated in a classical way to increase diversity. This approach is more likely to succeed because differences between the accepted inputs and the reference grammar often lies in human error, interpretation and relaxation of the grammar rules which is a part of what GMUTATOR attempts. Other mutation they attempt is adding alternatives to terminals and non-terminals. This mutation is also likely to succeed because it leads to token replacement and often language extensions follow the logic of pre-existing rules but with different keywords. This is especially relevant to us as they also mutate the generated inputs so new keywords are likely to be discovered and lead to accepted inputs. However this might not be sufficient as finding new keywords by chance suffers from the same unlikely probability problems as generating a correct invalid input.

RQ4: Is grammar mutation able to find significant non-standard features supported by parsers?

To answer this question we will compare the log files produced by a reference parser and the parser of the tool-under-test. We will consider the answer to RQ3 positive if grammar mutation produces an input which fulfils one of the following conditions:

(1) The input is rejected by the reference parser but accepted by the tool's parser.
(2) The input makes the tool's parser leak in its log file a grammar rule which is not compliant with the standard.
(3) The input contains a token which is rejected by the reference parser but accepted by the tool's parser, which can be detected by reading the log files.

2.2 Methodology

Our experiment will consist of two fuzzing campaigns, that we call acceptance testing and rejection testing. In each of these campaigns we will log all behaviour not conforming to the standard we encounter and classify them as non-compliance if the behaviour happens at the parsing stage. We will check whether the behaviour is acknowledged as a deviation from the standard in the official documentation, error message or code. If so the non-compliance will be considered "flagged as not supported", otherwise we will report the behaviour. Additionally we will record time taken by the campaigns except for the bug reduction time which will be reported separately to not give an unfair advantage to later experiments when comparing the different methodologies. We will also report on developer's willingness to fix the issues for all reported behaviours and the frequent root causes of non-compliances if identified.

In both these fuzz-testing campaigns, deciding whether parsing has been successful is complicated. One reason for that is most tools do not provide a simple way to run only their parser. This is especially true for closed-source tools for which there is no simple way to insert code to interrupt file processing after parsing. The other reason is that there might not be a clear-cut distinction between preprocessing, lexing, parsing and the rest of the processing done on the input. (We consider context-dependent checks such as definedness checks or type checking to not be part of parsing even if they are performed during the parsing phase.) Therefore the way we decide if a parser succeeds is one of the following:

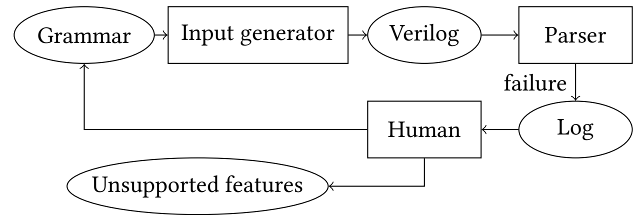

Figure 1: Acceptance testing overview

- If the tool is open-source then we modify it so that its return code indicates success when the parsing step is successful. This involves for instance disabling definedness checks and type checking performed at the parsing step, even if this prevents the tool from building an AST.
- If a tool is closed-source and its parser can be run on its own then we do the following. If the return code of that parser indicates succeeds then parsing succeded. Otherwise the same process as the one applied to closed-source tools without standalone parsers is applied.
- If a tool is closed-source and its parser cannot be run on its own, then a human inspects the log file and decides whether the failure is due to the parsing step or other processing. We do so without previous filtering or preprocessing because only failure could be filtered and in case of failure the log needs to be examined anyways to determine the root cause.

Acceptance testing is similar to a standard fuzzing campaign. It is explained in Section 2.3 and illustrated in Figure 1. Its first goal is to under-approximate the input grammar of the tool-under-test, that is to describe all standard compliant inputs that the tool-under-test accepts. To do that we find all standard compliant inputs the tool is supposed to accept but rejects, and remove the part of a standard compliant grammar describing them. After this fuzzing campaign we will have a grammar describing the maximal subset of the standard which is accepted by each tool. We will try to use these grammars to deduce a common subset of Verilog supported by all the tools and to make a list of common non-compliances. The subset will be useful for the second goal of acceptance testing. The second goal is to create a benchmark made of minimal test files similarly to those of SV-TESTS to test which features are supported by each tool. These test files will be minimal in the sense that the grammar-coverage of each file should be minimal, and features outside the subset supported by all tool should be avoided unless they are strictly necessary. We intend to contribute these files to SV-TESTS to make them publicly available and so that the anonymised tools' benchmark can be reproduced.

Rejection testing is similar to the work of Bendrissou et al. on Grammar Mutation [10]. It is explained in Section 2.4 and illustrated in Figure 2. Its goal is to find accepted inputs that the tool-under-test is supposed to reject. This campaign has to be run after acceptance testing because their technique grows the set of inputs described by the grammar, so it has to start from an under-approximation.

Table 1: Tools to test

Tool name	Tool category	Source availability
ANTLR4 SystemVerilog [8]	Grammar	Open-source
ANTLR4 Verilog [9]	Grammar	Open-source
ANTLR4_SystemVerilog_Parser [15]	Grammar	Open-source
Conformal	Equivalence checker	Proprietary (Cadence)
Diamond	FPGA synthesiser	Proprietary (Lattice)
Formalpro	Equivalence checker	Proprietary (Siemens)
Genus	ASIC synthesiser	Proprietary (Cadence)
hdlConvertor [1]	Converter between several HDL	Open-source
Icarus Verilog [35]	Simulator	Open-source
Jasper	Property checker	Proprietary (Cadence)
Leonardo Spectrum	Generic synthesiser	Proprietary (Siemens)
Moore [30]	Compiler	Open-source
Oasys-RTL	ASIC synthesiser	Proprietary (Siemens)
Quartus	FPGA synthesiser	Proprietary (Intel/Altera)
Questasim/Modelsim	Simulator	Proprietary (Siemens)
Surelog [12]	Parser	Open-source
Slang [27]	Parser	Open-source
sv-parser [16]	Parsing library	Open-source
sv2v [32]	SystemVerilog to Verilog converter	Open-source
Tree-sitter-verilog [11]	tree-sitter syntax file	Open-source
Verible [23]	Multitool	Open-source
Verific	Parser	Proprietary
Verilator [33]	Simulator	Open-source
Verismith [19]	Parser & Fuzzer	Open-source
Vivado	FPGA synthesiser	Proprietary (AMD/Xilinx)
Xcelium Parallel Logic Simulation	simulator	Proprietary (Cadence)
Yosys [31]	Generic synthesiser	Open-source

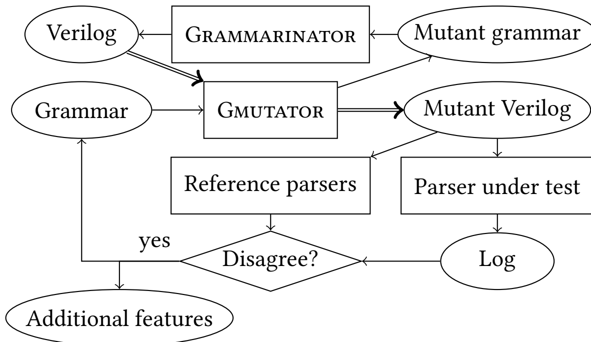

Figure 2: Rejection testing overview

We plan to test all tools listed in Table 1 unless we encounter licensing issues. Most of these tools expect SystemVerilog as input but some provide a Verilog mode that we will use. Indeed the fuzz-testing campaigns will be based on the IEEE 1364-2005 version of the standard but without the compiler directives that a preprocessing step can eliminate. We chose this version because:

- it is a well-defined subset with an associated standard that we can refer to,
- it is still widely used and relevant [13] even if superseded,
- it is almost a subset of the latest IEEE 1800-2023 [4] standard[3], and
- its size still allows a single person to write a fully compliant grammar within a reasonable time frame.

Inputs exposing a parser non-compliance in both fuzzing campaigns are used to identify non-compliances in the tools-under-test. This identification is done by a human using error message guidance, source code inspection and input file reduction. We reduce inputs either by using C-Reduce [28] with a script to reject uninteresting reduction candidates early, or by using Perses [34] with the grammar used to generate the input. When using C-Reduce, uninteresting candidates are rejected if we fail to parse them with the grammar from which the failed test was generated or fail to parse them with the Verismith parser, our fully compliant parser. This helps speeding up the reduction with closed source programs by avoiding their startup and licence checking time.

2.3 Acceptance Testing

The acceptance testing part of our experiment will be similar to standard fuzz-testing campaigns. A fuzz-testing pass will consist

[3]There are exceptions such as the removed supply0 and supply1 wire types, and planned for depreciation defparam statements.

of the following steps illustrated in Figure 1: A Verilog input is generated for each Verilog-consuming tool using the Verismith fuzzer, GRAMMARINATOR [20], a state of the art grammar based fuzzer, and TRIBBLE [18], a grammar-based systematic testing input generator based on k-path coverage. The generated inputs are passed to various Verilog consuming tools. The test fails if the parser fails. Failed tests are checked by a human to determine what unsupported feature caused the test to fail. This is a standard part of any fuzzing campaign which claims to report bugs and can be tackled as explained at the end of Section 2.2. The unsupported features are logged and disabled for the specific tool not supporting them in the input generation tools to prevent their generation in the next passes. This is also a standard part of black-box fuzzing because fuzzer tend to trigger the same few bugs until the generation parameters are changed. In our case, disabling a feature can be done by changing the configuration of probability distributions and changing the grammar used for generation. If some context awareness is needed, it can be added in Verismith and some may be added in GRAMMARINATOR. In case it cannot be done in GRAMMARINATOR, and no grammar change can be used to work around the unsupported feature, GRAMMARINATOR will be disabled for the following passes. This fuzzing campaign will continue until no test fails or until a feature cannot be disabled and further testing cannot continue without it being disabled.

One reason to use a bespoke tool like Verismith is the flexibility offered by a familiar code base. This familiarity allows us to quickly modify the source code. We may modify the source code to work around bugs and non-compliances, and to test new ideas. One idea tested in Verismith is the use of different random distribution for the choice made during random generation. Another advantage of a bespoke tool is to implement tooling to help the fuzzing without having to consider how the tool would have to work on general grammars. We expect this to be useful to increase coverage to a similar level to what is achievable by TRIBBLE.

2.4 Rejection Testing

The rejection testing part of our experiment will use the grammar mutation technique from Bendrissou et al. [10]. A fuzz-testing pass will consist of the following steps illustrated in Figure 2: For all tools, mutant grammars are generated by GMUTATOR [10]. These grammars are used by GRAMMARINATOR to generate big enough inputs in sufficient number to try to exercise the mutated parts. The inputs are passed to various Verilog-consuming tools. It is also passed to a reference Verilog parser and several SystemVerilog parsers to know whether it is supposed to be accepted or rejected. A test is failed depending on which parsers accepted and rejected the input. For each failed test the failure cause is investigated and logged.

A test accepted by a tool can fall into three categories: valid Verilog, valid SystemVerilog, invalid SystemVerilog. We are interested in invalid SystemVerilog inputs accepted by the tools, and valid SystemVerilog if the tool provides a Verilog mode. To discriminate tests in the first category we run a preprocessor then the Verismith parser. Discriminating between the second and third category is more complicated as no tool is fully compliant with SystemVerilog and all of the tool we run are also tested. We therefore do differential-testing with two of the most compliant SystemVerilog parsers we have available. The first parser is Verific, the most compliant industrial SystemVerilog parser. We chose it because it seems to be the most compliant SystemVerilog parser we know of. The second parser is ANTLR with the open-source ANTLR SystemVerilog Grammar. We chose it because it covers the full language and is the easiest to modify. The ease to modify matters to be able to quickly improve compliance when both reference parsers disagree. If they both accept the input the input is considered valid SystemVerilog, if they both reject the input the input is considered invalid SystemVerilog, if they disagree a human is used to decide after examining the log produced by all the parsers.

Table 2: Pilot experiment short results

Tool name	Fuzzing status
Conformal	Abandoned (wrong setup)
Formalpro	3 non-compliances and 1 bug
Icarus Verilog	5 non-compliance + 2 flagged as not supported
Jasper	Abandoned (uninformative log file)
Leonardo Spectrum	4 non-compliances
Oasys-RTL	Abandoned (uninformative log file)
Quartus	2 non-compliances
Questasim/Modelsim	5 non-compliances
Surelog	5 non-compliances
Slang	3 non-compliances
sv2v	3 non-compliances
Verible	3 non-compliances
Verific	Abandoned (uninformative log file)
Verilator	13 non-compliances
Vivado	4 non-compliances
Yosys	2 non-compliances flagged as not supported

3 Pilot Experiment

We ran a pilot experiment to prepare the grounds and test the viability of our planned experiment. This pilot experiment used the same methodology as acceptance testing with several key differences:

(1) inputs are generated only with the Verismith fuzzer,
(2) we tested a restricted set of tools listed in Table 2,
(3) open-source tools were not modified to stop after parsing, and
(4) we used only C-Reduce to reduce test cases.

The reason we only used Verismith is that we already implemented the grammar fuzzing part to test its parser and make sure it is fully-compliant. The reasons we tested a limited set of tools are that we did not yet gather a full list of tools, we did not install all of them, and we did not want to put unnecessary effort if the experiment would yield uninteresting results. The reason we did not modify open source tools is because we did not want to put the effort if the experiment would yield uninteresting results.

The goals we tried to achieve with this pilot experiment were:

(1) prepare an infrastructure to fuzz SystemVerilog-consuming tools,
(2) test the compliance of some open-source Verilog grammars,
(3) ensure grammar-based fuzzing is able to find non-compliance, and
(4) ensure the Verismith parser, AST and fuzzer are fully-compliant.

The steps to run this fuzzing fuzzing campaign and how we approached them are described in the rest of this section.

3.1 Preliminary Work

To prepare for the pilot experiment we had some preliminary work on Verismith. We implemented its new Verilog parser, prettyprinter, and AST with the goal of them being fully compliant.

The Verilog parser is able to parse all valid Verilog and reject all invalid Verilog in strict mode. It is also able to accept some invalid but commonly considered valid Verilog outside strict mode but we will not be using it in our experiments. The parser produces an AST which encodes all language constructs described in the input without encoding which way these language constructs are expressed in the output. For instance there are two way to express module ports in Verilog but the AST type unifies both so the parser does not record which one is used in the input.

The AST type is exactly able to encode all language constructs described in the standard and no more. The prettyprinter is able to print all AST, and by doing so it only produces valid Verilog. These last two features are relevant for the pilot experiment as the way out tool generates inputs is by randomly walking its AST type and printing the generated AST. The random walk is controlled by probability distributions, which are exposed as configuration options. The probability distribution available provide flexibility through several distribution families like geometric and Poisson distributions.

Walking recursive AST node types can lead to non-termination if not handled properly. To avoid non-termination and limit the size of the generated input, instead of opting for a hard depth limit, we opted for an attenuation factor which reduces the weight of non-terminals. This attenuation factor is updated by a user-set constant to the power of the number of children nodes. This ensures that deeper recursive AST become exponentially unlikely, so it limits depth like a hard depth limit. But unlike a hard depth limit it also limits width by ensuring that wider recursive AST become exponentially unlikely.

The prettyprinter cannot print every valid Verilog. In particular the prettyprinter cannot produce the compiler directives which can be eliminated by a preprocessing step and therefore the generated inputs should not exercise the tool-under-test's preprocessor. To add variety to the prettyprinter output some command line flags are available and, when different ways of printing an AST node are available, the one which conditions are the most restrictive and satisfied is chosen.

To disable the generation of features we provide a configuration to change the probability distribution of the random walk in the AST type. If the configuration is not available we can also modify the code to add and expose the necessary configuration. On top of that the prettyprinter provides flags to circumvent bugs in the tools-under-test's parsers.

3.2 Setup

Before being able to run the pilot experiment, some implementation work was necessary to setup the tools and infrastructure. The setup for the tools consisted of installing all the tools, ensuring their license are valid and accepted, setting-up an environment in which they can run, writing a script to run their parser on a file, and extracting an informative log file. The infrastructure coordinates the execution of a fuzz testing pass while solving several challenges:

(1) Process should be parallelised to save time, especially license checking time.
(2) Parallelisation of that many tools requires limitation of resources such as concurrent processes and run-time of a single process.
(3) Some of these tools do not allow several instances to run at the same time so number of concurrent instances of the same tool is also a resource to manage.
(4) Some processes can fail in a non fatal way, we should be able to restart them.
(5) Our servers crashed many times in the weeks prior to the experiment, so we should have checkpoints.
(6) All programs may not run on the same machine, so we should be able to interrupt a run and continue it on a different machine.

Algorithm 1: Process forest scheduling

```
Input: forest, resc
running ← ∅
to_retry ← ∅
repeat
    foreach ⟨proc, tree⟩ ∈ running do
        if proc is done then
            running ← running \ {⟨proc, tree⟩}
            resc ← resc + resources(root(tree))
            if return_code(proc) is special then
                to_retry ← to_retry ∪ {tree}
            else if return_code(proc) is success then
                forest ←
                  forest ∪ on_success(tree) ∪ on_end(tree)
            else
                forest ←
                  forest ∪ on_failure(tree) ∪ on_end(tree)
    checkpoint(running, forest, to_retry)
    foreach tree ∈ forest do
        if resources(root(tree)) ≤ resc then
            forest ← forest \ {tree}
            running ← running ∪
              {⟨launch(process(root(tree))), tree⟩}
            resc ← resc − resources(root(tree))
    wait_until_a_process_finishes(running)
until running = ∅
```

Our solution to tackle these challenges is to express the tasks to perform in a fuzzing pass as a forest of rooted trees. The algorithm

we use to schedule and run the forest is similar to Algorithm 1. The nodes of the trees are annotated with a process to run, the resources necessary to run the process, and an optional timeout. A node can have arbitrarily many children classified into three categories: run on success, run on failure, run on end. This way of expressing the fuzzing pass allows isolating critical parts requiring limited or potentially unavailable resources, like an exclusive tool license preventing several instances to run or a specific version of libgcc, from the parts which can be freely run and parallelised. Therefore this tackles the first three challenges.

The fourth challenge is tackled by filtering special return codes and adding the tree associated to the process of interest to a list of trees to retry later. The fifth challenge is tackled by serialising[4] to a file the forest left to run, the currently running trees, and the trees to retry. The sixth challenge is tackled by changing the available resources depending on which machine the algorithm is run and by exiting when no process can be run anymore.

3.3 Applying Changes for the Next Pass

After each tool is run on a single input, a log file is recorded and all other files are cleaned up. These log files are examined to determine whether parsing was successful. If parsing failed the log file is examined to list potential parsing errors. Each parsing error is investigated in isolation to determine a root cause. This investigation is helped with reducers and source-code examination. Reduction candidates are validated by the Verismith parser to ensure compliance and avoid startup time before being passed to the tool-under-test if the parsing succeeded. The reduced files are manually checked for compliance as the root cause of the failure could lie in Verismith not being fully-compliant. After the root cause of the error under investigation is determined, we either fix Verismith or are add an entry explaining the error to the list of non-compliances. The list of non-compliances also records if there was some developer acknowledgement of the feature not being supported either in the error message or in the source code.

The list of non-compliance is then used to disable the generation of further test that would fail for the same reason. This can be done because we identified the root cause of the issue and because the fuzzing tool we use provides means to do it. There are several ways our fuzzer allows disabling the generation of some files:

The first one is by setting to 0 the probability associated to the generation of specific AST nodes. This method is very coarse grained as the probability distribution are usually shared between all occurrences of an AST node type. If changing the probability distribution would disable the generation of files that would not trigger the error then we try the second way. The second way is to duplicate code and expose the necessary configuration to set the probability of generating the AST node of concern to 0. This method is fine grained but as the AST is not a concrete syntax tree, some language constructs cannot be disabled this way and we have to try our third option. The third way of disabling the generation of some inputs is by adding a flag to change the code emitted by the prettyprinter and working around the problematic generation when the flag is present.

[4]Processes are specified as command lines, so they can be easily serialised.

We did not encounter non-compliances which required more than these three methods to prevent triggering. We did find a bug which requires context awareness to not be triggered. Adding some context awareness can be done by changing the AST generation algorithm. And if all previous methods fail, we can process and/or filter inputs after their generation but before they are given to the tool-under-test.

3.4 Preliminary Results

The results are summarised in Table 2. This table reports if the tool didn't get tested and the reason, or a number of bugs and non-compliances and how many of these non-compliances were self-reported as non supported (as defined in Section 3.3). We had to give up testing some tools because the obtained log files were missing (Conformal) or did not allow us to pinpoint the unsupported feature with our methodology (Jasper, Oasys-RTL, Verific).

A non-compliance represents a family of failing inputs which according to the standard should be accepted. The counting of these non-compliance is somewhat arbitrary as a family of inputs can be arbitrary, but we tried to make these family focus on a part of a rule in the Verilog BNF which can be distinguished from other non-compliances. For instance the rule used by Surelog's parser in the example in the introduction contains two non-compliances. The first one is explained in the introduction. The second one is that an escaped identifier should be able to be followed by a space character, a tabulation character, a newline character or a fromfeed character; whereas this rule only accepts space characters. These two non-compliances are considered distinct because they are about different parts of the same rule. The first one is about the character class to which the Kleene star is applied, and the second one is about the character(s) after the Kleene star.

As expected most tools do not support all features from a 20 years old standard. According to licensing terms we cannot publish benchmark results for the closed source tools so we will not report individual results. The most common non-compliances we found were about escaped identifiers. Of these non-compliances, the two common were the Kleene star replaced by a Kleene plus and the preprocessor replacing what it considered a comment or a compiler directives inside escaped identifiers.

These results points toward the ability of fuzzing to find non-compliances in parsers. However some engineering work is still needed to proceed with our planned experiment. This engineering work includes installing, interfacing and getting licenses for all the missing tools, fixing and adapting the only Verilog grammar we found to use with Grammarinator and Gmutator, and interfacing each open-source tool with a compatible coverage-guided fuzzer.

Aside from the fuzzing campaign, we only found one candidate grammar for Verilog and this candidate is not compliant either. We did not find that out by running our fuzzer but by reading the grammar itself. This read revealed that the grammar assumes all compiler directives can be eliminated by a processing step which is not true: The standard specifies that some of compiler directives do not impact the semantics, but also allows tools have a different (unspecified) behaviour if they are used. We will therefore have to adapt this grammar before running our planned experiment.

4 Related Work

Our work relies on several previous work: Acceptance testing relies Grammarinator for input generation but we could have used any other black-box grammar-based fuzzer [14, 17, 20, 21] and on tribble [18] for the systematic input generation based on k-path coverage. Grammarinator was chosen because it is considered a state of the art grammar-based fuzzer and it is able to work with ANTLR grammars. tribble was chosen because k-path coverage is more general than production coverage, which is 1-path coverage. Even though tribble does not support ANTLR grammar, we will try to use it because we found bugs in our preliminary experiment that systematic testing based on production coverage would be unable to find. Rejection testing relies on input mutation from Gmutator [10].

The closest work to ours is Sv-Tests [6], a project by CHIPS Alliance, which benchmarks many open-source tools according to which features of the SystemVerilog standard they support. These tests are come from several sources: some are extracted from the language reference manual, some are taken from open-source tools' test suites, some are open-source hardware designs, and some are contributed by users. A great amount of manual work is required to categorise them, minimise them, and ensure their compliance. On top of that they lack any kind of systematic testing which is our novelty compared to this work. In particular we were able to quickly identify bugs that they missed with our pilot experiment, even for the most compliant parser they feature.

Sv-Tests tests only a limited set of tools, all open-source, and they do not attempt to exercise or list additional non-standard features. In some contexts, non-standard features are not perceived as an issue, but they do hinder design portability. On the other hand, if there are widely-supported non-standard features, then their identification can impact on future standardisation efforts.

Other SystemVerilog sources which can be used to perform feature testing include:

- SystemVerilog benchmark suites [5, 7, 25]. They are usually realistic or challenging files which test what people actually use and exercise the preprocessor. However they are not always standard compliant and use macros which values are stored in an external and non portable manner. This prevents these benchmark suites from being used for feature testing.
- Test files from open source tools. They are minimised to test specific features in a controlled manner. They usually cover more features at the expense of having a single test for each feature. They are not always standard compliant.
- Existing Verilog fuzzers [19, 36]. The existing ones are designed to fuzz features that the tools are able to parse because finding parsing bugs is not their focus. Therefore they are mostly useless for us.

To automate testing with existing inputs we need to know what part of the SystemVerilog grammar they cover so a compliant SystemVerilog grammar is still necessary. Grammar coverage give an approximation the real inputs covered, but the gap can be closed by existing mutation fuzzing tools. Even without mutation, we think these sources are a valuable way of finding parser bugs and compliance issues and used them to that end on a previous version of the Verismith parser.

There are other work similar to ours but not focused on Verilog. Kim et al. [22] build an input grammar by intercepting calls to C string manipulation functions. Their grammar is way simpler than the Verilog grammar and most tool use parser generator which may not rely on string manipulation functions.

5 Discussion

We think rejection testing can be improved by adding a feature in one of the tools we use. The main problem to overcome with rejection testing is that generating slightly invalid but accepted inputs is unlikely. We chose to use Gmutator because it should focus on producing likely accepted invalid inputs. To generate these inputs it first generates a mutant grammar, but then there is no mechanism to ensure that files produced with this grammar could not have been generated with the original grammar. Such a mechanism would avoid the generation of inputs which acceptance is already known, which would save time and increase fuzzing efficiency.

This feature could especially be useful because Grammarinator is unlikely to reach a mutated rule under several uses of the Kleene operator. The way Kleene operator are handled leads the number of repetitions to follow a geometric distribution. The production of 0 repetitions is more likely than any other number of repetition which prevents the generation of an input featuring the repeated rule. A rule under n uses of the Kleene operator will be reached with probability $(1-p)^n$ where p is the probability of stopping generation at 0 repetitions. A workaround would be choosing a small p at the expense of producing huge inputs because the expected number of repetition is $\frac{1}{p}$. This issue could solved by choosing other distributions for Kleene operators such as a Poisson distribution. The Verismith fuzzer provides the ability to choose the probability distribution of Kleene operators and many more things between varied families of distributions.

The process of growing an under-approximation of the accepted inputs could also be improved by using the messages in the log files produced by parsers. Instead of relying on acceptance of an input which is unlikely with mutations, messages such as "At line X, column Y: invalid token Z" and "At line X, column Y: Expected tokenA, tokenB, ..." can be exploited to know whether a mutation is interesting. Even though they all indicate a failure, the position of the failure is of prime interest because these parsers usually process inputs in only one direction[5]. All the mutations that happened before the point of failure were therefore successful. On top of that token lists like in the second example message can be exploited to discover new tokens which can be added to a dictionary. This could be used for black-box fuzzing to improve automation provided token position from the parser can be exploited which can be complicated because of tabulation characters. In the cases exploiting these error messages also require editing the parser's code then they would have no interest because it is no longer black-box fuzzing, and coverage-guided techniques will likely be better.

[5] with potential backtracking

Full automation of feature testing might be achievable using coverage-guided fuzzing. It is not a black-box fuzzing technique so it is not applicable to closed source tools. Testing coverage-guided fuzzing is still interesting as an experiment to increase automation of the technique as much as possible. It would involve changing the code of the tool-under-test to isolate the parsing step. Input on which the tool's parser fails but a fully-compliant parser succeeds are inputs we are searching for in acceptance testing, while inputs on which the tool's parser succeeds but a fully-compliant parser fails are inputs we are searching for in rejection testing. If from these inputs we can edit a grammar to make it describe what the tools accepts then we consider that feature testing can indeed be automated fully with coverage-guided fuzzing. We know there is work on deducing a grammar from a set of inputs but we do not know if there is work on editing a approximation of a grammar given a set of inputs.

A lot of effort put in our experiment goes into dealing with the quirks of black-box fuzzing on closed-source tools with restrictive licensing. We think there is still a lot of possible improvements given enough engineering time. One area of improvement is the glue between the fuzzing script and the tools. We settled with running tools in parallel to save on license checking time, but a better solution might have been writing a software server which starts the tool and performs an action to force a license check. This software server would then run the tool on provided inputs while ensuring it stays gets to a proper state before processing the next input. This is feasible and could even be worth on unreliable physical servers like ours which crash from time to time and need these software servers to be restarted.

Another area of improvement is the adequacy of closed-source parser with our definition of parsing. Even though their license forbid reverse engineering, a judgement of the Court of Justice of the European Union [26] clarifies that for the purpose of fixing bugs and interoperability we can reverse engineer and modify the code of these tools. So we are allowed to insert breakpoints in binaries and disable functions like we would do for open-source tools[6][7]. This would allows greater automation of our technique, and other previously non black-box fuzzing methodology to be significantly improved.

Acknowledgments

We thank Europractice for providing tools and licenses, Michalis Pardalos for installing some of the tested tools, Bachir Bendrissou for helping us with using Grammarinator, Alastair Donaldson for suggesting this conference and helping fleshing out the angle used to structure this article, and the reviewers for giving valuable feedback and suggestions such as the comparison to systematic testing.

References

[1] [n. d.]. hdlConvertor. Retrieved June 20, 2024 from https://github.com/Nic30/hdlConvertor

[2] 1996. IEEE Standard Hardware Description Language Based on the Verilog(R) Hardware Description Language. *IEEE Std 1364-1995* (1996), 1–688. https://doi.org/10.1109/IEEESTD.1996.81542

[3] 2006. IEEE Standard for Verilog Hardware Description Language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* (2006), 1–590. https://doi.org/10.1109/IEEESTD.2006.99495

[4] 2024. IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. *IEEE Std 1800-2023 (Revision of IEEE Std 1800-2017)* (2024), 1–1354. https://doi.org/10.1109/IEEESTD.2024.10458102

[5] Christoph Albrecht. 2005. IWLS 2005 Benchmarks. (2005). https://iwls.org/iwls2005/benchmark_presentation.pdf

[6] CHIPS Alliance. [n. d.]. SystemVerilog Report. Retrieved May 24, 2024 from https://chipsalliance.github.io/sv-tests-results/

[7] Luca Amarù, Pierre-Emmanuel Gaillardon, and Giovanni De Micheli. 2015. The EPFL Combinational Benchmark Suite. *Proceedings of the 24th International Workshop on Logic & Synthesis (IWLS)*. http://infoscience.epfl.ch/record/207551

[8] Mustafa Said Ağca. 2022. ANTLR4 SystemVerilog grammar. Retrieved June 20, 2024 from https://github.com/antlr/grammars-v4/tree/master/verilog/systemverilog

[9] Mustafa Said Ağca. 2022. ANTLR4 Verilog grammar. Retrieved June 20, 2024 from https://github.com/antlr/grammars-v4/tree/master/verilog/verilog

[10] Bachir Bendrissou, Cristian Cadar, and Alastair F. Donaldson. 2023. Grammar Mutation for Testing Input Parsers (Registered Report). In *Proceedings of the 2nd International Fuzzing Workshop* (Seattle, WA, USA) *(FUZZING 2023)*. Association for Computing Machinery, New York, NY, USA, 3–11. https://doi.org/10.1145/3605157.3605170

[11] Aliaksei Chapyzhenka. [n. d.]. Tree-sitter-verilog. Retrieved June 20, 2024 from https://github.com/tree-sitter/tree-sitter-verilog

[12] Alain Dargelas and Henner Zeller. 2020. Universal Hardware Data Model *(WOSET 2020)*. https://woset-workshop.github.io/WOSET2020.html#article-10

[13] Harry D. Foster. 2022. *The 2022 Wilson Research Group Functional Verification Study*. Technical Report. https://blogs.sw.siemens.com/verificationhorizons/2022/10/10/prologue-the-2022-wilson-research-group-functional-verification-study

[14] Andrew Fryer, Thomas Dean, and Brian Lachine. 2023. Input Output Grammar Coverage in Fuzzing. In *MILCOM 2023 - 2023 IEEE Military Communications Conference (MILCOM)*. 937–943. https://doi.org/10.1109/MILCOM58377.2023.10356308

[15] Miguel Guerrero. [n. d.]. ANTLR4_SystemVerilog_Parser. Retrieved June 20, 2024 from https://github.com/miguel-guerrero/antlr4_system_verilog_parser

[16] Naoya Hatta. [n. d.]. Sv-parser. Retrieved June 20, 2024 from https://github.com/dalance/sv-parser

[17] Nikolas Havrikov and Andreas Zeller. 2019. Systematically Covering Input Structure. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 189–199. https://doi.org/10.1109/ASE.2019.00027

[18] Nikolas Havrikov and Andreas Zeller. 2020. Systematically covering input structure. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 189–199. https://doi.org/10.1109/ASE.2019.00027

[19] Yann Herklotz and John Wickerson. 2020. Finding and Understanding Bugs in FPGA Synthesis Tools. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Seaside, CA, USA) *(FPGA '20)*. Association for Computing Machinery, New York, NY, USA, 277–287. https://doi.org/10.1145/3373087.3375310

[20] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2018. Grammarinator: a grammar-based open source fuzzer. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation* (Lake Buena Vista, FL, USA) *(A-TEST 2018)*. Association for Computing Machinery, New York, NY, USA, 45–48. https://doi.org/10.1145/3278186.3278193

[21] Hossein Keramati and Seyed-Hassan Mirian-Hosseinabadi. 2015. Generating semantically valid test inputs using constrained input grammars. *Information and Software Technology* 57 (2015), 204–216. https://doi.org/10.1016/j.infsof.2014.09.007

[22] Su Yong Kim, Sungdeok Cha, and Doo-Hwan Bae. 2013. Automatic and lightweight grammar generation for fuzz testing. *Computers & Security* 36 (2013), 1–11. https://doi.org/10.1016/j.cose.2013.02.001

[23] Google LLC, David Fang, Sergey Sokolov, Jonathan Mayer, Jeremy Colebrook-Soucie, Cameron Korzecke, Carissa Kathuria, and Henner Zeller. [n. d.]. Verible. Retrieved June 20, 2024 from https://chipsalliance.github.io/verible

[24] Andreas Lööw and Magnus O. Myreen. 2019. A Proof-Producing Translator for Verilog Development in HOL. In *2019 IEEE/ACM 7th International Conference on Formal Methods in Software Engineering (FormaliSE)*. 99–108. https://doi.org/10.1109/FormaliSE.2019.00020

[25] Kevin E. Murray, Scott Whitty, Suya Liu, Jason Luu, and Vaughn Betz. 2015. Timing-Driven Titan: Enabling Large Benchmarks and Exploring the Gap between Academic and Commercial CAD. *ACM Trans. Reconfigurable Technol. Syst.* 8, 2, Article 10 (mar 2015), 18 pages. https://doi.org/10.1145/2629579

[6] However bypassing license checks might also not be allowed even with this ruling.
[7] This is not legal advice. We are not lawyers. This document has not been reviewed by a lawyer.

[26] Court of Justice of the European Union. 2021. Case C-13/20: Judgment of the Court (Fifth Chamber) of 6 October 2021 (request for a preliminary ruling from the Cour d'appel de Bruxelles — Belgium) — Top System SA v Belgian State (Reference for a preliminary ruling — Copyright and related rights — Legal protection of computer programs — Directive 91/250/EEC — Article 5 — Exceptions to the restricted acts — Acts necessary to enable the lawful purchaser to correct errors — Concept — Article 6 — Decompilation — Conditions). https://eur-lex.europa.eu/legal-content/en/TXT/?uri=CELEX:62020CJ0013

[27] Michael Popoloski. [n. d.]. Slang. Retrieved June 20, 2024 from https://sv-lang.com

[28] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-Case Reduction for C Compiler Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 335–346. https://doi.org/10.1145/2254064.2254104

[29] Dave Rich. 2023. What's Next for SystemVerilog in the Upcoming IEEE 1800 standard *(DVCon 2023)*. https://dvcon-proceedings.org/document/whats-next-for-systemverilog-in-the-upcoming-ieee-1800-standard

[30] Fabian Schuiki, Andreas Kurth, Tobias Grosser, and Luca Benini. 2020. LLHD: a multi-level intermediate representation for hardware description languages. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 258–271. https://doi.org/10.1145/3385412.3386024

[31] David Shah, Eddie Hung, Claire Wolf, Serge Bazanski, Dan Gisselquist, and Miodrag Milanovic. 2019. Yosys+nextpnr: An Open Source Framework from Verilog to Bitstream for Commercial FPGAs. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. 1–4. https://doi.org/10.1109/FCCM.2019.00010

[32] Zachary Snow. [n. d.]. Sv2v. Retrieved June 20, 2024 from https://github.com/zachjs/sv2v

[33] Wilson Snyder. [n. d.]. Verilator. Retrieved June 20, 2024 from https://www.veripool.org/verilator

[34] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: Syntax-Guided Program Reduction. In *Proceedings of the 40th International Conference on Software Engineering*. Association for Computing Machinery, 361–371. https://doi.org/10.1145/3180155.3180236

[35] Stephen Williams. [n. d.]. Icarus Verilog. Retrieved June 20, 2024 from https://steveicarus.github.io/iverilog

[36] Claire Wolf. 2019. VlogHammer. Retrieved May 24, 2024 from https://github.com/YosysHQ/VlogHammer

Received 2024-06-21; accepted 2024-07-22