An Open-Source Environment to Measure Coverage of Verilog Fuzzers

1st Rafael Fontes Sumitani
Computer Science Department
UFMG
Belo Horizonte, MG, Brazil
rafaelsumitani@dcc.ufmg.br

2nd Fernando M Quintão Pereira
Computer Science Department
UFMG
Belo Horizonte, MG, Brazil
fernando@dcc.ufmg.br

***Abstract*—Electronic Design Automation (EDA) tools are software applications used by engineers in the design, development, simulation, and verification of electronic systems and integrated circuits. These tools typically process specifications written in a Hardware Description Language (HDL), such as Verilog, SystemVerilog or VHDL. Thus, effective testing of these tools requires programs written in these languages. There are existing resources to provide such input, such as ChiGen, a probabilistic Verilog generator, and ChiBench, a curated suite of Verilog programs from open-source repositories. This study presents the development of an open-source experimental framework designed to evaluate the code coverage achieved by Verilog fuzzers and benchmark suites. The framework automates the execution of coverage experiments, while enabling extensibility and reproducibility. Experiments were conducted on widely used open-source EDA tools to assess the testing effectiveness of both ChiGen and ChiBench. Results demonstrate that ChiBench consistently achieves the highest coverage, while ChiGen outperforms other existing fuzzers. The study underscores the utility of ChiGen and ChiBench for testing EDA tools and highlights the framework's broader applicability to general code coverage experiments.**

***Index Terms*—code coverage, verilog, testing, fuzzing, benchmark**

I. Introduction

EDA (Electronic Design Automation) tools are specialized software applications used by engineers in the design, development, simulation, and verification of electronic systems and integrated circuits (ICs). Examples of such tools are Verilator, Icarus Verilog, Cadence Jasper Formal Verification Platform and Synopsys Design Compiler. EDA tools cover various stages of the electronic design process, from conceptualization and design entry to implementation, verification, and testing. Therefore, ensuring the reliability and correctness of EDA tools is of great importance, as Herklotz and Wickerson [1] note: "*[...] the final hardware is only as reliable as the logic synthesis tool that produces it.*"

All EDA tools operate on similar types of input data: programs in some Hardware Description Language (HDL), such as Verilog, SystemVerilog, VHDL or SystemC. Thus, effective testing of such tools requires programs in these languages. The problem of providing abundant HDL inputs has been tackled in previous work through two different approaches – ChiBench [2] and ChiGen [1].

ChiGen is a fuzzer which uses a probabilistic model to synthesize random Verilog programs. ChiGen's probabilistic model was built by observing programs from ChiBench, a benchmark suite of Verilog programs mined from open-source repositories, which can also be used by itself as input for testing. Both ChiGen and ChiBench have been used to test different EDA tools and were able to find a total of 9 bugs in open-source EDA tools, thus proving their efficiency for testing.

There are, however, other existing benchmark suites and fuzzing tools that provide Verilog programs for the purpose of testing EDA tools. Therefore, it would be in our interest to compare ChiGen and ChiBench with other existing solutions to find further evidence of their effectiveness as testing tools. There are different metrics that can be used to make such comparisons. The one that we have chosen is code coverage, both for lines and for branches.

To effectively compare different Verilog test suites, we need an experimental framework capable of efficiently obtaining code coverage values and comparing results across program sets. Furthermore, this framework should be extensible, enabling the integration of additional EDA tools and program benchmarks as they become available. Thus, the main contribution of this work is the development of such environment, that can be used to validate the effectiveness of ChiGen and ChiBench as testing tools.

II. Background

A. Related Work

1) Verilog Fuzzers: The term "fuzz" was first used by Miller et al. [3], where they generated random inputs to test the reliability of UNIX command line utilities. The core idea behind fuzzing is to automatically generate random, and often unexpected, inputs to test a target program to find bugs, vulnerabilities, or unexpected behaviors.

ChiGen, the tool implemented in our earlier work, is a Verilog fuzzing tool based on probabilistic context-free grammars (PCFGs). In summary, we have built a grammar which

[1]Both available at https://github.com/lac-dcc/chimera

assigns probabilities to production rules. These probabilities have been computed by observing a large collection of Verilog programs, ChiBench. ChiGen also tries to embed context in its probabilities by using n-grams. Inspired by the concept of n-grams from Natural Language Processing [4], ChiGen's probabilities can be set to consider sequences of n production rules, instead of relying solely on the current rule. By using PCFGs and n-grams, we can synthesize programs that are random, but are still similar to real world Verilog designs.

After synthesizing a valid Verilog syntax tree, ChiGen applies different semantic analyses, such as type inference and liveness analysis, before actually outputting the code in text format. Figure 1 displays an example of a Verilog program synthesized by ChiGen.

```
01 module module_1 (
02   id_1,
03   id_2
04 );
05   inout wire id_2;
06   inout wire id_1;
07   id_3(
08     .id_0(1)
09   );
10   reg id_4;
11   reg id_5;
12   assign id_4 = 1;
13   supply0 id_6 = (1);
14   module_0 modCall_1 ();
15   always @(negedge id_6) id_5 <= 1;
16   assign id_4 = ~id_5;
17   initial begin : LABEL_0
18     #1 id_4 <= 1;
19   end
20 endmodule
```

Fig. 1: Snippet of a Verilog program synthesized by ChiGen.

Along with ChiGen, there are other fuzzers that produce Verilog designs, such as Verismith [1], LegoHDL [5], TransFuzz [6], and VlogHammer [7]. Verismith, for instance, is a Verilog fuzzer inspired by Csmith [8], and it synthesizes programs based on probabilities, in a similar manner to ChiGen. However, Verismith considers a limited subset of Verilog's grammar, and it cannot synthesize programs with more complicated language constructs, such as modules and primitive gates instantiations. Moreover, Verismith differs from ChiGen in that it uses SMT solvers to verify that the netlist synthesized by the tool under test is formally equivalent to the original design, which adds a higher level of testing rigor. Therefore, although Verismith synthesizes simpler programs, it has more complex testing and equivalence checking capabilities.

Recent advancements in Artificial Intelligence have also introduced tools based on Large Language Models (LLMs) for the synthesis of Verilog designs. VeriGen [9], VerilogReader [10], and CraftRTL [11] are examples of such models. VeriGen, for instance, is a fine-tuned version of the CodeGen-16B model that can synthesize syntactically valid Verilog code from prompts. Although the authors highlight the capabilities of their model to solve simple tasks, such as implementing Verilog designs for education exercises, but they do not go into detail regarding the usage of their model to synthesize input for testing. This fact hinders direct comparison with fuzzing tools, which are the focus of this work.

2) Benchmark suites: A *Benchmark Suite* is a collection of programs used to test computing systems that process such programs. There exist open source benchmark collections tailored for EDA tools, such as the ISCAS Benchmark Circuits [12] (31 circuits), the EPFL Combinational Benchmark Suite [13] (23 circuits), the RAW Benchmark Suite [14] (12 programs), the KOIOS collection (19 circuits implementing different neural networks) and the Titan23 suite of 23 circuits [15]. However, these collections contain a small number of programs: typically less than 50. This fact is unfortunate because, in the words of Wang and O'Boyle [16]: *"Although there are numerous benchmark sites publicly available, the number of programs available is relatively sparse compared to the number that a typical compiler will encounter in its lifetime."*

The absence of large benchmark suites targeted for EDA tools was addressed in our previous work with the introduction of ChiBench [2]. ChiBench is a curated program collection which consists of over 50,000 Verilog designs mined from open-source repositories on GitHub. Programs included in ChiBench go through an automated filtering process, which guarantees that they are syntactically valid and adhere to key semantic rules, such as the consistency of port directions. All programs were taken from repositories with permissive licenses (e.g., MIT License), which allow for public use and distribution. The scale and diversity of ChiBench make it uniquely suited for a range of applications, such as training Large Language Models (LLMs) and direct testing of EDA tools.

B. Code Coverage

Code coverage is a metric used in software testing to measure the percentage of a code base that is exercised when a program is tested by a particular test suite. A test with high code coverage will have a higher chance of having bugs detected, since a large part of the program's source code is executed. Currently, code coverage is a standard metric for testing quality in the software industry, and it is adopted by large companies such as Google [17]. Given the importance of code coverage as a metric for testing quality, this work focuses on coverage analysis to evaluate the effectiveness of ChiGen and ChiBench for testing EDA tools when compared to other test suites.

In order to measure the percentage of the source code that is executed, different *coverage criteria* may be considered. For instance, line coverage, which is one of the most adopted criteria, defines coverage in terms of the percentage of lines that are executed during testing. Similarly, branch coverage

assesses the ratio of control flow branches that are exercised during program execution, providing deeper insights into what paths of the program have been properly tested.

There are many existing tools for measuring code coverage, which target programs in different programming languages. GNU's gcov and Clang's source-based code coverage are alternatives for code coverage measurement in C/C++ programs. Most EDA tools considered in this work are implemented in C/C++, therefore we shall consider coverage tools targeted for these languages.

C. EDA Tools

Electronic Design Automation (EDA) tools are used in various steps of the design of electronic systems. There are numerous tools, both closed-source and open-source, that are used for important tasks such as simulation (e.g., Cadence Xcelium [18] and Verilator [19]), formal verification (e.g., Cadence Jasper [20] and SymbiYosys [21]), and equivalence checking (e.g., Synopsys Formality [22] and EQY [23]). Thus, such tools are of paramount importance to hardware design, and ensuring the correctness and reliability of EDA tools directly contributes to the overall reliability of the final hardware.

In this work, we primarily focus on testing open-source EDA tools. Since we employ coverage testing, we need to be able to compile our target tools from source using some code coverage tool, as mentioned in Section II-B. Therefore, we can only do these experiments if we have access to the tool's source code. However, our methodology can be easily adapted to test closed-source EDA tools by anyone who has access to the source code.

III. Methodology

In this work we employed coverage testing for four different EDA tools. Table I displays all used tools, along with their versions. Furthermore, in addition to ChiGen, three other fuzzers were used to generate designs for testing. Table II describes fuzzers that were used in our experiments.

EDA tool	Version
Verible's syntactic analyzer [24]	Commit 75c38da (2025-01-07)
Verible's code formatter [25]	Commit 75c38da (2025-01-07)
Icarus Verilog's parser [26]	Commit 30123f8 (2025-01-09)
Verilator [19]	Commit 052812b (2025-01-09)

Table I: EDA tools used in coverage experiments

Fuzzer	Version
Verismith [1]	Commit 5077809 (2024-11-14)
TransFuzz [6]	Commit 9305895 (2024-11-13)
VlogHammer [7]	Commit 8c828f3 (2019-01-15)

Table II: Verilog fuzzers used in coverage experiments

ChiBench was the only Verilog benchmark suite that was included in our experiments. As mentioned in Section II-A2, large suites of Verilog designs for testing are scarce. Therefore, including any test suite with less than 10,000 programs would not lead to a fair comparison against both ChiBench and the fuzzers, since using fewer programs is more likely to result in lower code coverage.

To build program collections with all available fuzzers, we opted to randomly generate 10,000 Verilog designs using each fuzzer. VlogHammer was the only exception since it has the limitation of only synthesizing 3061 designs. Additionally, ChiGen has a parameter which determines the size of its context for probabilities (n-grams). Therefore, we also ran experiments with 6 different versions of ChiGen, using $n = 1, 2, 3, 4, 5, 6$. Due to ChiBench's large size, we used a random sample of 10,000 designs, instead of using all 50,000. Random sampling was chosen to ensure a representative subset while maintaining consistency with other fuzzers' program counts.

To obtain code coverage values, all EDA tools were compiled using Clang's source based coverage to profile their binaries for coverage. Clang's coverage was used due to its ease of use both for compiling programs and for producing coverage reports, which can be generated using llvm-cov. The version of LLVM used in our experiments was 16.0.0.

Our experiments then consisted of running each EDA tool using all available programs sets as inputs. When obtaining coverage for collections of programs, we collected coverage for each individual program, but we also retained coverage data from designs that were run previously. Thus, we were able to analyze how code coverage progressed when adding more designs to the experiment, and we also obtained the total coverage achieved by each collection. After collecting coverage data for all pairs of EDA tools and program collections, we can finally draw conclusions and analyses regarding the effectiveness of ChiGen and ChiBench as testing tools.

IV. Experimental Framework for Coverage Testing

As mentioned in Section I, one of the primary goals of this work is to produce an experimental environment that ensures both reproducibility and extensibility. Therefore, we had to develop an infrastructure that would support this.

In order to facilitate the execution and the extension of our coverage experiments, we implemented a Python script which automates our experimental pipeline. This script calls a separate Shell script that runs our target tools using all Verilog designs from all collections, as described in Section III, while saving all intermediate results in CSV format. Once all designs are run for a tool, the Python script then produces a chart that summarizes line and branch coverage values for that tool.

Moreover, to improve extensibility, our Python script takes a YAML [27] configuration file which informs all EDA tools and collections that must be used in experiments. This YAML file also allows users of the script to customize other aspects of the run, such as path information for the binaries of EDA tools and visual details of the generated charts. The YAML format was chosen due to its readability and ease of use, as well as for its integration with Python.

Figure 2 displays an example of how our YAML configuration file works. The *datasets* sequence specifies which

program collections will be used in our experiments. Each item in *datasets* identifies a collection and configures certain aspects, such as its name, its local path, and how it should be represented in the final charts. On the other hand, the *tools* sequence specifies EDA tools that should be used in coverage experiments, along with details such as their local paths.

Our Python script also allows the user to customize the Shell script that will run the target tools using the specified program collections and collect coverage data. This not only helps cover any unique requirements of running each tool, but we also believe that it allows the usage of our experimental framework for code coverage experiments that are not restricted to EDA tools. After all, the only requirement is that this separate Shell script runs the desired tool using the specified input and then produces coverage data in the correct CSV format. Other than that, it is not limited to EDA tools.

```
datasets:
  - label: "1gram"
    name: "1 gram"
    path: "../chigen_programs/1gram"
    line_color: "tab:blue"
    line_style: "solid"
  - label: "chibench"
    name: "ChiBench"
    path: "../sample_database"
    line_color: "r"
    line_style: "dashed"
tools:
  - label: "iverilog"
    name: "Icarus Verilog's parser"
    binary_path: "../../iverilog/ivl"
  - label: "verilator"
    name: "Verilator"
    binary_path: "../../verilator/bin/
```

Fig. 2: Snippet from the YAML file that can be used to configure coverage experiments.

Our experimental framework has, however, many different dependencies, some of which can be difficult to set up. For instance, there may be dependencies needed to build our target EDA tools. Thus, we have also provided a Dockerfile, that creates a Docker image with all necessary dependencies to simplify reproducing our experiments. This Dockerfile also uses Shell scripts to automate the process of compiling our target EDA tools for code coverage with Clang, which can be a cumbersome process. The complete experimental infrastructure, with both the Dockerfile and the Python script, is open-source and available at https://github.com/lac-dcc/chimera/tree/main/coverage.

V. Results

In this Section we will display and analyze the results of our coverage experiments. The utilized version of ChiGen for these experiments was Commit ffd99e4 (2024-12-19). A complete description of the other fuzzers that were used is available at Table II. Furthermore, a full description of all EDA tools that were used in our experiments is available at Table I. Results are presented as charts where the y-axis is the code coverage percentage and the x-axis is the number of programs used. For each EDA tool there are two charts – one for branch coverage and another for line coverage.

Experiments were conducted on a Linux machine featuring Ubuntu version 22.04 LTS, equipped with an AMD Ryzen 9 5900X 12-Core with a clock of 3.7GHz, and 32 GB of RAM (DDR4).

A. Verible's syntactic analyzer

Figure 3 reports code coverage values for Verible's syntactic analyzer. ChiBench, the only benchmark suite with human-written programs in our experiment, achieved the highest coverage both for branches (44%) and for lines (35%). ChiGen did not fall behind by a lot, with all its versions achieving more than 39% branch coverage and more than 25% line coverage. ChiGen's version with 1 gram context achieved the highest coverage, whereas 2 gram and 3 gram were the versions with the lowest coverage, but the margin of difference was small. Verismith, VlogHammer and TransFuzz achieved lower coverage percentages when compared to all versions of ChiGen, with less than 33% branch coverage and 15% line coverage. It is also evident that these fuzzers were not able to increase their coverage values when adding more designs, as opposed to what happened for ChiBench and ChiGen.

B. Verible's code formatter

Figure 4 displays results for Verible's code formatter. Results for this tool differed from those of Verible's syntactic analyzer. ChiBench still achieves the highest coverage values, both for branches and for lines, at nearly 40%. However, ChiGen is proportionally closer to ChiBench's branch coverage than it was in the previous experiment. The same pattern was observed for the other three fuzzers. Nevertheless, Verismith, VlogHammer and TransFuzz still presented lower code coverage than ChiGen.

C. Icarus Verilog's parser

Figure 5 shows the obtained results when experimenting with Icarus Verilog's parser. The dominance of ChiBench still prevailed, at around 40% for both coverage criteria. ChiGen also remained the fuzzer with the largest code coverage, at nearly 30%. However, TransFuzz presented a much lower coverage both for branches and for lines, and was not able to stay close to Verismith, which was a pattern that appeared in the experiments for other EDA tools. Another difference is that ChiGen's version with 6 gram was the version with the largest code coverage.

D. Verilator

Figure 6 illustrates results for coverage experiments with Verilator. The relative order between tools stayed the same in

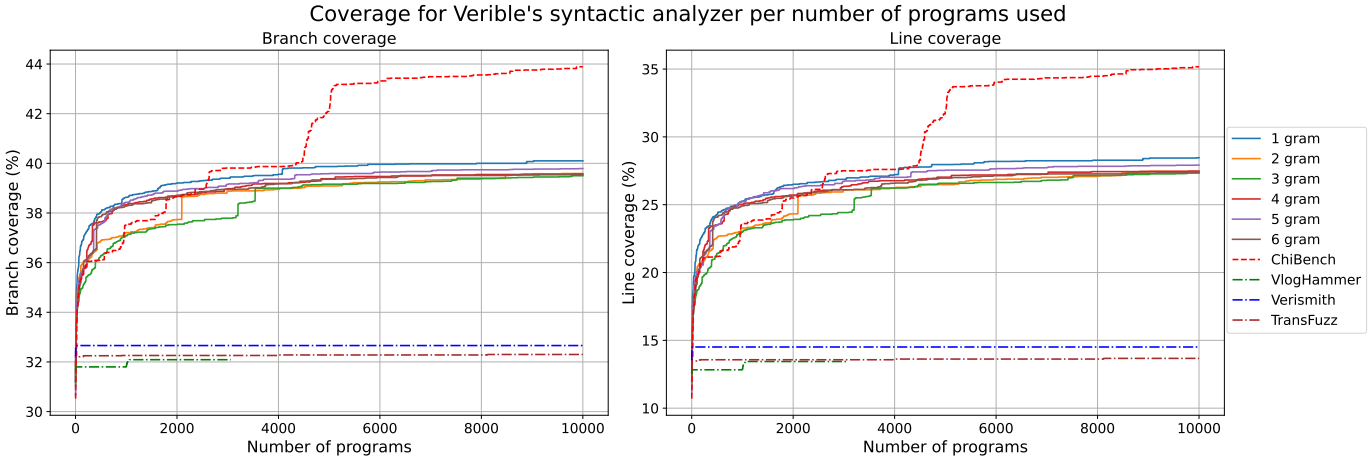

Fig. 3: Code coverage results for Verible's syntactic analyzer.

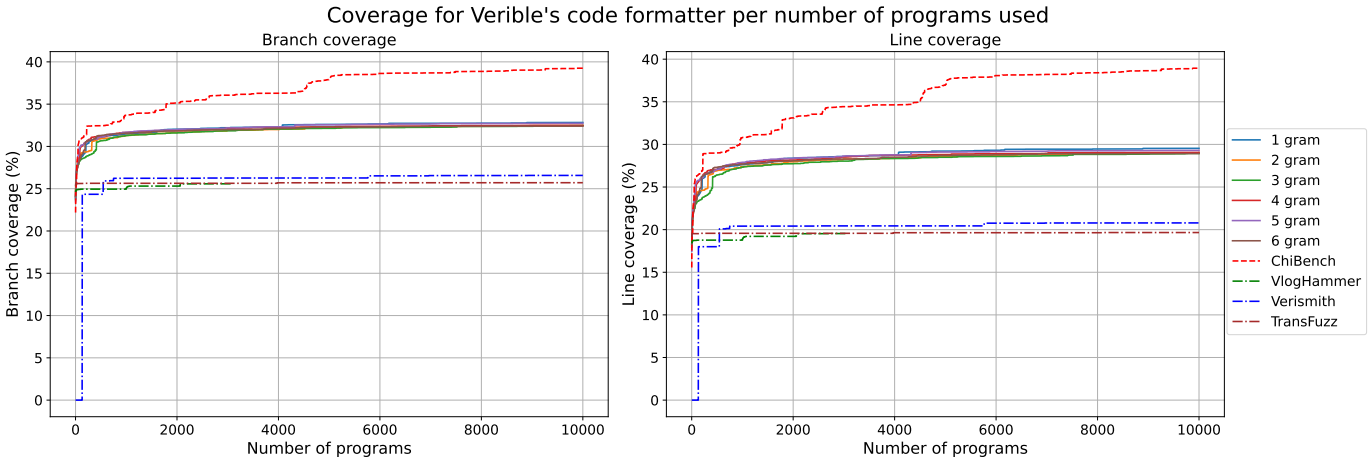

Fig. 4: Code coverage results for Verible's code formatter.

this experiment; ChiBench first, then ChiGen, then all other fuzzers. However, Verismith, VlogHammer, and TransFuzz performed better in this experiment than they did in the previous ones. Up until around 1,000 programs used, Verismith was even able to surpass ChiGen. Nonetheless, ChiGen continued to improve its coverage by adding more designs whereas Verismith and TransFuzz plateaued and were not able to achieve higher code coverage.

E. Final Remarks

The results across all four EDA tools reaffirm the effectiveness of ChiBench and ChiGen as testing tools. Both achieve higher code coverage values when compared to other tools, which indicates that a larger percentage of the target tools are being exercised. Thus, the chance for finding bugs or unexpected behaviors is also higher.

ChiBench achieved the largest code coverage for all experiments, indicating that it is a very viable collection to be used for coverage experiments, with a diverse population of designs. It is important to note that these experiments were performed with a subset of ChiBench, which contains around 20% of its actual size. Therefore, performing coverage experiments with the entirety of ChiBench might lead to even higher coverage.

Regarding the performance for Verilog fuzzers, ChiGen consistently outperforms other fuzzers, both for simple tools like Verible's code formatter, but also for more complex tools such as Verilator. However, it was not possible to observe any clear superiority among versions of ChiGen with different context sizes, which may indicate that this parameter is not too important for code coverage. One interesting observation was that Verismith, TransFuzz and VlogHammer reach code coverage plateaus and are not able to increase their coverage by adding more designs to the test. This may indicate that code synthesized by ChiGen is more diverse than the code generated by these other fuzzers and, therefore, they will exercise different parts of our target EDA tools. This observation could be validated further by analyzing other metrics, such as the number of unique tokens in each collection. However, such analyses are not in the scope of this work.

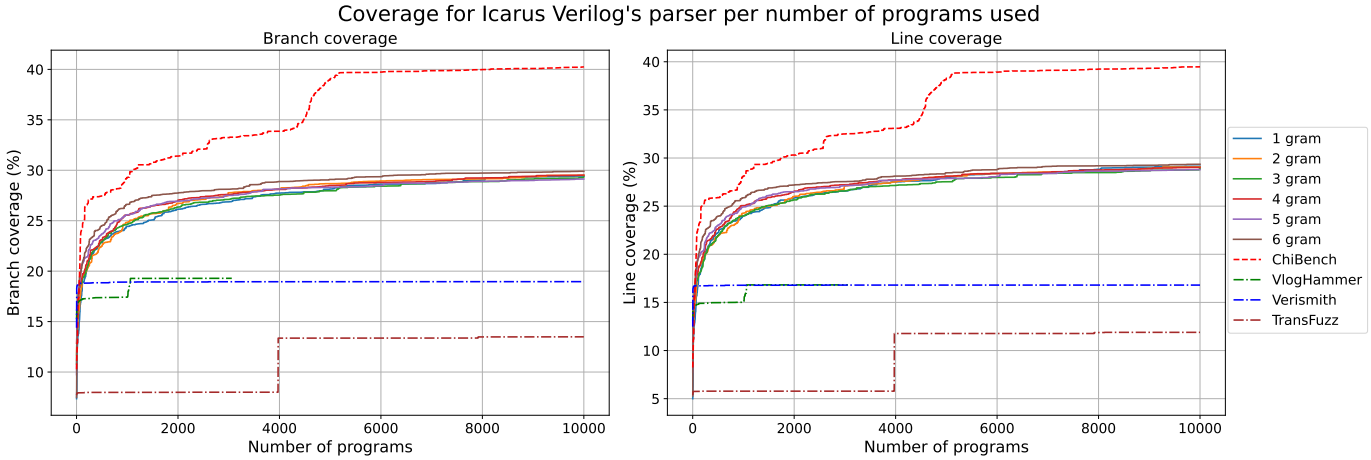

Fig. 5: Code coverage results for Icarus Verilog's parser.

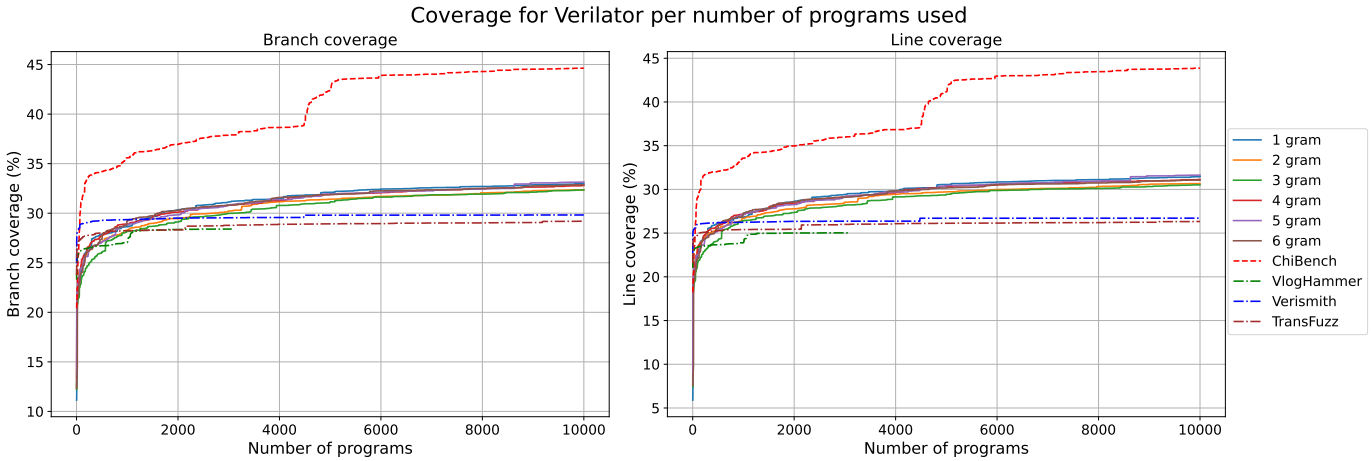

Fig. 6: Code coverage results for Verilator.

VI. Future Work

There is still room for improvement in this work, both in our experimental framework and in the experiments themselves. One first improvement is related to the performance of experiments. Running many different EDA tools with different large collections of designs may take several hours – or even days. It is quite challenging to optimize experiments for a single tool, since the coverage for one design depends on all previous designs that were run. However, experiments for different EDA tools are completely independent. Therefore, one possible way of improving performance would be to update our Python script to run experiments for different tools in parallel. It is important to note that running these tools and collecting coverage may be very CPU-intensive. Thus, parallelization should also take this factor into account in order to avoid overloading the CPU.

There are also improvements that can be made for our experiments. For instance, we were not able to find large benchmark suites of Verilog designs, and thus only ChiBench was included in our experiments. One possibility for finding more collections of human made Verilog code is to look for corpora of designs that have been used to train LLMs that generate Verilog code, such as VeriGen [9]. These collections are not specifically designed for testing, but they could be included in our coverage experiments to enrich the results. Furthermore, another possible enhancement would be to augment our experiments by using different criteria for code coverage (e.g., function coverage) or by combining different collections, such as ChiBench and ChiGen, to attempt an increase in code coverage.

VII. Conclusion

This work has described an open-source experimental framework for testing code coverage of Verilog fuzzers and benchmark suites. With its aid, we provided further evidence supporting the usefulness of ChiGen and ChiBench – tools developed in our previous work – as resources for testing EDA tools. Furthermore, due to the extensible nature of this experimental framework, it can be used to test the effectiveness of different test suites via code coverage for different kinds of target tools, and it is not limited to EDA. All scripts developed as part of this study are publicly available at

https://github.com/lac-dcc/chimera, along with ChiGen and ChiBench.

Acknowledgment

Rafael Sumitani thanks João Victor Amorim, Luiza de Melo, Raissa Maciel, Augusto Mafra, and Mirlaine Crepalde for their substantial contributions to this project.

References

[1] Y. Herklotz and J. Wickerson, "Finding and understanding bugs in FPGA synthesis tools," in *ACM/SIGDA Int. Symp. on Field-Programmable Gate Arrays*, ser. FPGA '20. ACM, 2020.

[2] R. Sumitani, J. V. Amorim, A. Mafra, M. Crepalde, and F. M. Q. Pereira, "Chibench: a benchmark suite for testing electronic design automation tools," 2024. [Online]. Available: https://arxiv.org/abs/2406.06550

[3] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, no. 12, p. 32–44, dec 1990. [Online]. Available: https://doi.org/10.1145/96267.96279

[4] D. Jurafsky and J. H. Martin, *Speech and Language Processing (2nd Edition)*. USA: Prentice-Hall, Inc., 2009.

[5] Z. Xu, S. Guo, G. Zhao, P. Zou, X. Li, and H. Jiang, "A novel hdl code generator for effectively testing fpga logic synthesis compilers," 2024. [Online]. Available: https://arxiv.org/abs/2407.12037

[6] F. Solt and K. Razavi, "Lost in Translation: Enabling Confused Deputy Attacks on EDA Software with TransFuzz," in *USENIX Security*, Aug. 2025. [Online]. Available: Paper=https://comsec.ethz.ch/wp-content/files/mirtl_sec25.pdfURL=https://comsec.ethz.ch/mirtl

[7] YosysHQ. Vloghammer. "Accessed: 2025-01-20". [Online]. Available: https://yosyshq.net/yosys/vloghammer.html

[8] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 283–294. [Online]. Available: https://doi.org/10.1145/1993498.1993532

[9] S. Thakur, B. Ahmad, H. Pearce, B. Tan, B. Dolan-Gavitt, R. Karri, and S. Garg, "Verigen: A large language model for verilog code generation," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 29, no. 3, apr 2024. [Online]. Available: https://doi.org/10.1145/3643681

[10] R. Ma, Y. Yang, Z. Liu, J. Zhang, M. Li, J. Huang, and G. Luo, "Verilogreader: Llm-aided hardware test generation," 2024. [Online]. Available: https://arxiv.org/abs/2406.04373

[11] M. Liu, Y.-D. Tsai, W. Zhou, and H. Ren, "Craftrtl: High-quality synthetic data generation for verilog code models with correct-by-construction non-textual representations and targeted code repair," 2024. [Online]. Available: https://arxiv.org/abs/2409.12993

[12] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profiles of sequential benchmark circuits," in *ISCAS*. New York, USA: IEEE, 1989, pp. 1929–1934.

[13] L. Amaru, P.-E. Gaillardon, E. Testa, and G. D. Micheli, "The epfl combinational benchmark suite," Feb. 2019. [Online]. Available: https://doi.org/10.5281/zenodo.2572934

[14] J. Babb, M. Frank, V. Lee, E. Waingold, R. Barua, M. Taylor, J. Kim, S. Devabhaktuni, and A. Agarwal, "The raw benchmark suite: computation structures for general purpose computing," in *FCCM*. USA: IEEE Computer Society, 1997, p. 134.

[15] K. E. Murray, S. Whitty, S. Liu, J. Luu, and V. Betz, "Timing-driven titan: Enabling large benchmarks and exploring the gap between academic and commercial cad," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 8, no. 2, mar 2015. [Online]. Available: https://doi.org/10.1145/2629579

[16] Z. Wang and M. O'Boyle, "Machine learning in compiler optimisation," 2018.

[17] M. Ivanković, G. Petrović, R. Just, and G. Fraser, "Code coverage at google," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 955–963. [Online]. Available: https://doi.org/10.1145/3338906.3340459

[18] Cadence Design Systems, Inc. Xcelium logic simulator. "Accessed: 2025-01-19". [Online]. Available: https://www.cadence.com/en_US/home/tools/system-design-and-verification/simulation-and-testbench-verification/xcelium-simulator.html

[19] W. Snyder, *Verilator Manual*, 2024, accessed: 2025-01-19. [Online]. Available: https://veripool.org/ftp/verilator_doc.pdf

[20] Cadence Design Systems, Inc. Jasper formal verification platform. "Accessed: 2025-01-19". [Online]. Available: https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification.html

[21] YosysHQ. Symbiyosys (sby) documentation. "Accessed: 2025-01-19". [Online]. Available: https://symbiyosys.readthedocs.io/en/latest/

[22] Synopsys, Inc., *Formality: Equivalence Checking and Interactive ECO*, "Accessed: 2025-01-19". [Online]. Available: https://www.synopsys.com/content/dam/synopsys/implementation&signoff/datasheets/formality-and-formality-ultra-ds.pdf

[23] YosysHQ. Equivalence checking with yosys (eqy) documentation. "Accessed: 2025-01-19". [Online]. Available: https://yosyshq.readthedocs.io/projects/eqy/en/latest/

[24] CHIPS Alliance. verible-verilog-syntax. "Accessed:

2025-01-20". [Online]. Available: https://chipsalliance.github.io/verible/verilog_syntax.html
[25] ——. verible-verilog-format. "Accessed: 2025-01-20". [Online]. Available: https://chipsalliance.github.io/verible/verilog_format.html
[26] Stephen Williams. Ivl - the core compiler. "Accessed: 2025-01-20". [Online]. Available: https://steveicarus.github.io/iverilog/developer/guide/ivl/index.html
[27] The YAML Project. The official yaml web site. "Accessed: 2025-01-20". [Online]. Available: https://yaml.org/