Probabilistic Synthesis of Verilog Programs to Test EDA Tools

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. This work presents ChiBench, a curated benchmark suite comprising more than 50K programs mined from open-source repositories. Additionally, this work also introduces ChiGen, a tool which synthesizes Verilog programs from scratch based on a probabilistic language model, thereby increasing the number of available inputs for testing.**

***Index Terms*—benchmark, verilog, testing, program synthesis**

I. Introduction

EDA (Electronic Design Automation) tools are software applications used by engineers in the design, development, simulation, and verification of electronic systems and integrated circuits (ICs). EDA tools cover various stages of the electronic design process, from conceptualization and design entry to implementation, verification, and testing. These tools operate on similar types of input data: programs in some Hardware Description Language (HDL), such as Verilog, SystemVerilog, VHDL or SystemC. Thus, the effective development and testing of such tools require programs in these languages.

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 [1] (31 circuits), the EPFL Combinational Benchmark Suite [2] (23 circuits), the RAW Benchmark Suite [3] (12 programs), the KOIOS collection (19 circuits implementing different neural networks) and the Titan23 suite of 23 circuits [4]. 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 [5]: *"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."*

This work mitigates the aforementioned problem with two different contributions. First we introduce ChiBench, an open collection of 50K Verilog programs, mined from open-source GitHub repositories. ChiBench is a much larger collection of programs when compared to existing benchmark suites. It has also been filtered using existing EDA tools, in order to select programs which are syntactically and semantically valid.

Along with ChiBench, we also introduce ChiGen, a tool which can synthesize Verilog programs based on an n-gram probabilistic language model. Thus, ChiGen can be used to generate realistic programs from scratch and increase the number of available inputs for testing EDA tools.

II. Related Work

Random synthesis of inputs for testing has existed for at least three decades [6]. The generation of HDL programs, however, is a topic which has gained more popularity with the rise of large language models (LLMs) [7, 8, 9]. Although LLMs excel at generating syntactically and semantically valid code, they are more usually applied in the context of code completion. Thus, instead of generating a random Verilog program from scratch, they can produce test cases for an existing design [8], or even complete a Verilog module based on its specification [7]. Nevertheless, the automated generation of unseen random Verilog programs for testing, which ChiGen aims to do, is still a problem which has not been solved to the best of our knowledge.

We have already mentioned in Section I the lack of extensive benchmark suites formed by hardware specification languages. However, this scenario has also seen changes in recent years due to the rising popularity of LLMs. As an example of this new trend, at the end of 2023, Thakur et al. [7] released VeriGen, a fine-tuned version of CodeGen [10] for the synthesis of Verilog specifications. In the process of tuning CodeGen, Thakur et al. have collected 50K Verilog circuits. However, in contrast to ChiBench, the dataset used by Thakur et al. has not undergone any form of filtering; hence, we do not know if these programs are semantically valid. Neither they are publicly available, what hinders a direct comparison with ChiBench. Independent evaluations of VeriGen has found that "*A primary contributing factor to this shortfall [the inability to uncover bugs in EDA tool] is the insufficiency of HDL code resources for training*" [10].

III. A Curated Collection of Verilog Programs

A. Mining Open-Source Programs

In order to build our Benchmark Suite, we have mined programs from open-source GitHub repositories, using GitHub's REST API[1]. We use GitHub's API to build a list of candidate Verilog repositories. Said list is sorted by popularity (measured as the number of stargazers). We remove from the candidate list repositories that are not available for public usage, due to the lack of a license. Thus, for each repository R in the sorted list, we have implemented a Python script that proceeds as follows:

1) Clone R and locally copy all its `.v` files;
2) Assign a unique name to each `.v` file, based on its repository, its local path, and the current number of programs in ChiBench;
3) Remove any special characters from the file's name to avoid encoding issues.

We repeat the above sequence of steps for all the repositories in the base list, until reaching a predefined number of files. This threshold is set upon calling the mining script.

B. Curating the Data

After we have copied the necessary number of Verilog files from GitHub, we proceed to select valid programs. To this effect, we only keep files that are syntactically and semantically valid. Thus, this process involves passing the files through two sieves. The first sieve, the syntax analysis, happens via the Verible's syntactic analyzer. Verible[2] is a suite of Verilog/SystemVerilog developer tools, which includes a parser and syntactic analyzer.

At this stage, if Verible's parser cannot build an abstract syntax tree for a file, we discard it. The program in Figure 1 would be filtered out by the syntactic filter. It contains a missing semicolon at Line 7. Such syntactically invalid files are uncommon in the mining process. Nevertheless, they occur, as the repositories contain, for instance, files that are still under development.

```
01 module counter (input clk, input rst,
02   output reg [7:0] data);
03   always @(posedge clk) begin
04     if (rst || data == 8'hff)
05       data <= 8'h00;
06     else
07       data <= data + 8'h01
08   end
09 endmodule
```

This program is syntatically invalid because it misses a semicolon at the end of line 7

Fig. 1. Specification filtered out by syntactic verification.

Once we remove any syntactically invalid programs, we use the semantic analyzer available in the Jasper Formal Verification Platform[3] to filter out any semantically invalid programs. Notice that Jasper's HDL analyzer also rejects invalid syntax. However, Jasper's analyzer is more computationally expensive than Verible's because it also considers semantic analysis, being more restricted to Verilog language standards. Consequently, in order to reduce the number of programs sent for semantic analysis, we chose to filter out syntactically invalid programs before using Jasper.

Figure 2 shows an example of a program that fails the semantic sieve due to a type inconsistency. In this case, the IEEE standard forbids the declaration of data ports with the wire type. Our data generation process considers each file independently, thus, this error might occur. Nevertheless, our experience is that most Verilog programs can be successfully validated as a single compilation unit.

```
01 module counter (
02   input clk,
03   input rst,
04   output wire [7:0] data
05 );
06   always @(posedge clk) begin
07     if (rst || data == 8'hff)
08       data <= 8'h00;
09     else
10       data <= data + 8'h01;
11   end
12 endmodule
```

This program is semantically invalid because the **data** port is declared with the **wire** type. However, the standard forbids using **wire** ports for procedural assignments (assignments within procedural blocks, like **always**)

Fig. 2. Verilog specification that fails the semantic test.

C. Use Cases

As a proof of concept on how to use such a benchmark suite, we have used ChiBench to test different EDA tools. Namely, we have tested Verible's parser and code obfuscator, Yosys[4], Verilator[5] and Icarus Verilog[6]. The methodology for these tests was simple – we used each program in the collection as input for each tool. Among programs which yielded non-zero exit codes, we searched for any crashes or unexpected behaviors. Under this methodology, we were able to find two bugs, which have already been reported:

- https://github.com/chipsalliance/verible/issues/2159: Verible's code obfuscator crashed when reading a program that only contains pragma directives. This bug has not been fixed as of July 2024.
- https://github.com/verilator/verilator/issues/5276: Verilator crashed when running for a program with long lines. This bug has also not been fixed as of July 2024.

Nevertheless, the usage of ChiBench is not limited to directly testing EDA tools. As mentioned in Section II, such

[1]Available at https://docs.github.com/en/rest

[2]Available at https://github.com/chipsalliance/verible

[3]Available at https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification.html

[4]Available at https://github.com/YosysHQ/yosys

[5]Available at https://github.com/verilator/verilator

[6]Available at https://github.com/steveicarus/iverilog

a large collection of programs may be invaluable for training machine learning models.

IV. Probabilistic Synthesis of Verilog Programs

A. Overview

ChiGen's main goal is to stochastically generate random Verilog programs in order to enrich the variety of inputs for testing EDA software. The tool was implemented using C++20, with `nlohmann/json`[7] and `cxxopts`[8] as dependencies. Figure 3 provides an overview of how it works.

The process for randomly producing a program was based on the concept of n-gram language models, which will be further explained in Section IV-B. Such a language model needs a large corpus of Verilog programs, from which it can estimate probabilities for each program construct, such as modules and `always` blocks. We have built ChiBench for this exact purpose.

With the probabilities of each construct in hand, all there is left is to synthesize the program. Programs generated by ChiGen are syntactically valid from their inception. However, there is still need to enforce semantic correctness. This is the role of semantic analysis, which is detailed in Section IV-E.

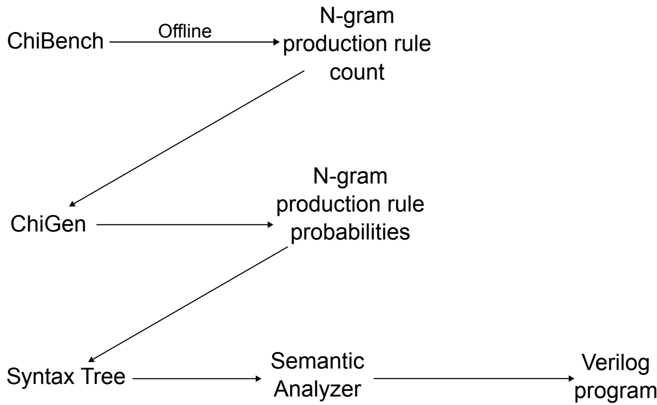

Fig. 3. An overview of ChiGen.

B. N-gram Language Models

In the context of Natural Language Processing, a language model assigns probabilities to upcoming words, or sequences of words, in a sentence [11]. An *n-gram language model* assumes that the probability of a given word is conditioned by the n previous words in the sequence. This concept is useful because it is a primitive way to model context: the likelihood of a word depends on the context which precedes it.

Therefore, we want to use this idea for synthesizing Verilog programs with the aim of producing more realistic programs. However, since we reason about programs, not sentences, we cannot deal with words. Instead, we assign probabilities to *production rules* in Verilog's grammar.

[7]Available at https://github.com/nlohmann/json
[8]Available at https://github.com/jarro2783/cxxopts

In order to better understand this idea, think about the syntax tree. Each production rule corresponds to a node in the program's syntax tree. Thus, ChiGen's language model synthesizes a Verilog syntax tree by assuming that the probability for the next production rule depends on its n closest ancestors in the tree. Figure 4 shows an example of how this idea works.

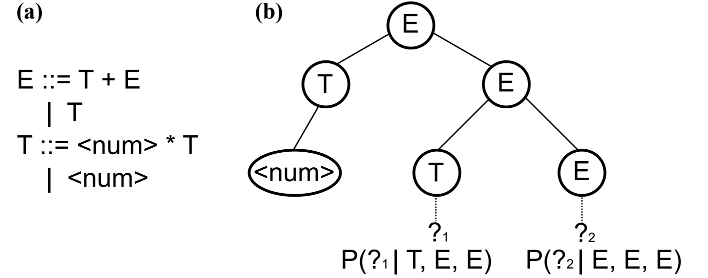

Fig. 4. (a) Simple grammar definition. (b) Example of the definition of probabilities for the next production rules in a 3-gram setting. Notice that $?_1$ and $?_2$ could be sequences of rules, instead of a single rule.

Based on this concept, we can use a large corpus to compute probabilities for each production rule in Verilog's grammar, and for a given context size. Section IV-C further explains the process for computing probabilities.

The computed probabilities can then be used with a simple algorithm to stochastically synthesize Verilog syntax trees. Algorithm 1 illustrates it.

Algorithm 1 Algorithm for synthesizing a syntax tree using n-grams

Input: `n`, probability distribution
Output: Head of synthesized syntax tree

```
1: head ← starting production rule
2: stack ← {head}
3: while stack ≠ ∅ do
4:   nextRule ← stack.pop()
5:   context ← getContext(nextRule,n)
6:   productions ← next production rules randomly
     chosen based on the probabilities for context
7:   for all production in productions do
8:     stack.push(production)
9:     nextRule.addChild(production)
10:  end for
11: end while
12: return head
```

C. Calculating Probabilities

In order to compute probabilities for production rules, we need a large corpus of Verilog programs, from which we can count the number of times we see each production rule. This will give a good estimate of how likely a production rule is in a given context. Although ChiBench has proven useful for other purposes, it was originally conceived for computing probabilities for ChiGen.

We have used Verible's parser as a means to obtain all the production rules which are generated for a program. This is possible by using the trace of Verible's parser, which outputs production rules in the order in which they are processed. This allows us to count how many times a rule appears and, since they appear in a well-defined order, we can also get the context of where a rule appears. Thus, we can count the number of times that each rule appears given the context of n preceeding rules.

Once we know how many times each production rule appears in a given context, the process for computing the probability for the next production rule given the current context is simple. For each possible context, we check what are the possible production rules and what are their counts. The count of a production rule then serves as a weight for the computation of a discrete probability distribution. Consequently, the sum of probabilities for the next possible production rules in a given context is 1.

One of the drawbacks of using the trace of Verible's parser is that it may be very long for large programs. Therefore, this process could take dozens of hours for a very large set of programs such as ChiBench. Fortunately, this process can be done offline, since we do not have to recount the occurrences of production rules in the dataset every time we want to synthesize a new program.

We have implemented a Python script which applies this idea of using Verible's parser to count the occurrences of production rules for a given program and a context size n. The script outputs a JSON file, which stores how many times a set of production rules appears in a specific context. After we have applied this script for every program in our dataset, the JSON file with the accumulated counts is then used as input for ChiGen, which will use it to compute probabilities as described earlier.

D. Building a Syntax Tree

Once ChiGen has computed probabilities using the input JSON file, it can synthesize a Verilog syntax tree using Algorithm 1. We have implemented Verilog's syntax tree in ChiGen's code, based on the production rules used by Verible's parser.

Notice that, because the nodes in the syntax tree are synthesized based on a valid Verilog grammar, it is impossible that a production rule is used where it is syntactically invalid. Therefore, the syntax trees synthesized by ChiGen are syntactically correct by definition.

E. Semantic Correctness

Although syntax trees synthesized by ChiGen are always syntactically correct, there is still need to enforce semantic correctness. After all, a syntactically valid program may contain semantic errors, as shown in Figure 2.

We have taken two different approaches with the purpose of increasing semantic correctness of synthesized programs. The first approach involved changes in Verible's grammar, so that some semantically invalid patterns would be rejected during parsing. This was applicable in two situations: to avoid mixed ANSI and Non-ANSI Port Declarations, and to avoid mixed named and positional port connections. Figure 5 shows an example of such a situation. Therefore, ChiGen works with a modified version of Verible's parser, which guarantees that it does not recognize such programs as syntactically valid.

```
01 module counter (
02   output reg [7:0] data,
03   clk,
04   rst
05 );
06  input clk;
07  input rst;
08  always @(posedge clk) begin
09    if (rst || data == 8'hff)
10      data <= 8'h00;
11    else
12      data <= data + 8'h01;
13    end
14 endmodule
```

This port declaration uses ANSI style

These ports use non-ANSI style

Fig. 5. The first port in this module uses ANSI style for port declarations, whereas the other two use non-ANSI style. The IEEE standard does not allow such mixing, and therefore this program is semantically invalid.

The second approach is applied directly to the synthesized syntax tree. This is necessary whenever we need to change values in the synthesized programs to make them semantically correct. For instance, the initially synthesized tree does not contain any valid identifiers or literals, only placeholders. Thus, we must replace these placeholders with actual values. We also have to ensure that an identifier is not used before it is declared, which must be taken into account when updating the tree. These semantic analyses and actions was implemented with the use of the Visitor design pattern [12], which allows to easily define operations to be applied to different subclasses in the tree structure.

Although these approaches increase the percentage of semantically valid programs we can synthesize, there is still much room for improvement. Achieving a 100% ratio of valid programs is very difficult when dealing with stochastically synthesized programs. However there are other kinds of analyses that which be implemented to enhance the semantic correctness of programs synthesized by ChiGen. For example, a type checker could greatly improve semantic correctness, by preventing the use of identifiers in contexts where their type is not allowed.

F. Use Cases

We have tested different EDA tools with programs synthesized by ChiGen to demonstrate its usefulness, as we did with ChiBench. Verible's code obfuscator, parser, and code formatter were included in this process. The methodology was similar to what we described in Section III-C – we used synthesized programs as inputs and looked for any crashes.

We have observed two bugs, and both have been reported to Verible's community:

- https://github.com/chipsalliance/verible/issues/2181: Verible's parser crashed instead of reporting syntax errors related to instantiation type. This bug has already been fixed in newer versions of Verible.
- https://github.com/chipsalliance/verible/issues/2189: Verible's code formatter crashed when dealing with a program with a variable of type `output logic signed`. This bug has not been fixed as of July 2024.

V. Conclusion

This paper has described two different approaches which aim to improve the availability of inputs for testing EDA tools. The first, ChiBench, is a curated collection of 50K Verilog programs mined from open-source repositories. The second, ChiGen, is a tool for probabilistic synthesis of Verilog programs based on an n-gram language model. In addition to explaining the methodology behind each approach, this paper has also demonstrated their usefulness by showing different bugs which were found in popular EDA tools using programs taken from ChiBench and ChiGen. Both ChiBench and ChiGen are publicly available at https://github.com/lac-dcc/chimera.

As mentioned in Section IV-E, there is still room for improvement in the semantic analyzer module of ChiGen. Thus, future work should involve developing new approaches for enforcing semantic correctness, such as a type analyzer.

Acknowledgment

This project is sponsored by Cadence Design Systems. Rafael Sumitani also thanks João Victor Amorim, Luiza de Melo, Augusto Mafra, and Mirlaine Crepalde for their incredible contributions to the project.

References

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

[2] 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

[3] 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.

[4] 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

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

[6] 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

[7] 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

[8] 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

[9] B. Nadimi and H. Zheng, "A multi-expert large language model architecture for verilog code generation," 2024. [Online]. Available: https://arxiv.org/abs/2404.08029

[10] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," 2023.

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

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995.