

**UNIVERSIDADE FEDERAL DE MINAS GERAIS**  
**Instituto de Ciências Exatas**  
**Programa de Pós-Graduação em Ciência da Computação**

João Victor Amorim Vieira

**Automatic Generation of Verilog Designs for Testing EDA Tools**

Belo Horizonte  
2025

João Victor Amorim Vieira

**Automatic Generation of Verilog Designs for Testing EDA Tools**

**Final Version**

Thesis presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Fernando Magno Quintão Pereira

Belo Horizonte  
2025

2025, João Victor Amorim Vieira.  
Todos os direitos reservados.

|       |                                                                                                                                                                                                                                                |
|-------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|       | Vieira, João Victor Amorim.                                                                                                                                                                                                                    |
| V665a | Automatic generation of verilog designs for testing EDA tools [recurso eletrônico] / João Victor Amorim Vieira. – 2025.                                                                                                                        |
|       | 1 recurso online (74 f. il.) : pdf.                                                                                                                                                                                                            |
|       | Orientador: Fernando Magno Quintão Pereira.                                                                                                                                                                                                    |
|       | Dissertação (mestrado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência Da Computação.                                                                                                            |
|       | Referências: f. 70-74.                                                                                                                                                                                                                         |
|       | 1. Computação - Teses. 2. Verilog (Linguagem descritiva de hardware). - Teses. I. Pereira, Fernando Magno Quintão. II. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação. III. Título. |
|       | CDU 519.6*17 (043)                                                                                                                                                                                                                             |

Ficha catalográfica elaborada por Célio Resende Diniz, bibliotecário CRB  
6/2403 - Universidade Federal de Minas Gerais – ICEX.



UNIVERSIDADE FEDERAL DE MINAS GERAIS  
INSTITUTO DE CIÊNCIAS EXATAS  
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

## FOLHA DE APROVAÇÃO

Automatic Generation of Verilog Designs for Testing EDA Tools

**JOÃO VICTOR AMORIM VIEIRA**

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

A handwritten signature in black ink, reading "Fernando M Q Pereira".

PROF. FERNANDO MAGNO QUINTÃO PEREIRA - Orientador  
Departamento de Ciência da Computação - UFMG

A handwritten signature in black ink, reading "Luis Humberto Rezende Barbosa".  
DR. LUIS HUMBERTO REZENDE BARBOSA  
Jasper - Cadence Design SystemsA handwritten signature in black ink, reading "Mateus Beck Rutzig".  
Prof. MATEUS BECK RUTZIG  
Departamento de Eletrônica e Computação - UFSMA handwritten signature in black ink, reading "John Wickerson".  
DR. JOHN WICKERSON  
Electrical and Electronic Engineering - Imperial College London

Belo Horizonte, 23 de maio de 2025.

*Dedico este trabalho aos meus pais, Fabiana e Geovan, ao meu irmão, Bernardo, e à minha namorada, Maria Fernanda. O apoio, o carinho e a confiança de vocês foram fundamentais em cada passo dessa jornada. Sem vocês, nada disso teria sido possível.*

# Acknowledgments

I would like to express my deepest gratitude to my advisor, Fernando Magno Quintão Pereira, for his incalculable guidance, support, and the many lessons shared throughout this journey. I am also sincerely thankful to Rafael Sumitani and Luiza de Melo for their direct contributions and unwavering support during the development of this project.

To my colleagues in the laboratory, thank you for the shared experiences, insightful discussions, and the many moments of learning along the way. I extend my appreciation to Cadence Design Systems for funding this project and providing the necessary resources to make it a reality.

Finally, I am grateful to UFMG and the Department of Computer Science for offering the environment and opportunities that made this work possible.

*“Even the smallest person can change the course of the future.”*  
(Tolkien, J.R.R. The Fellowship of the Ring.)

# Resumo

A validação de ferramentas de Automação de Projeto Eletrônico (EDA) depende do uso de benchmarks—projetos escritos em Linguagens de Descrição de Hardware (HDLs), como Verilog, SystemVerilog ou VHDL. Embora existam coleções de benchmarks para essas linguagens, elas são geralmente limitadas em tamanho e variedade. Essa escassez tem sido cada vez mais evidente, devido à crescente demanda por conjuntos de dados para o treinamento de grandes modelos de linguagem voltados para esse domínio. Para lidar com essa limitação, este trabalho apresenta uma metodologia e uma correspondente ferramenta, para a geração de projetos realistas em Verilog. A ferramenta, chamada ChiGen, foi originalmente desenvolvida para testar a plataforma de Verificação Formal Jasper<sup>®</sup>, um produto da Cadence Design Systems. Atualmente disponibilizada como software de código aberto, ChiGen demonstrou capacidade para identificar bugs inéditos (zero-day) em diversas ferramentas, incluindo Verible, Verilator e Yosys. Este artigo descreve os princípios que orientam o design do ChiGen, com foco em três aspectos principais: (i) geração orientada por gramáticas probabilísticas, (ii) inferência de tipos baseada no algoritmo de Hindley-Milner, e (iii) injeção de código viabilizada por análise de fluxo de dados. Quando executado em hardware convencional, o ChiGen supera ferramentas de fuzzing existentes para Verilog, como Verismith, TransFuzz e VlogHammer, em aspectos como diversidade estrutural, cobertura de código e capacidade de detecção de bugs.

**Palavras-chave:** Verilog, Síntese, Teste.

# Abstract

Testing Electronic Design Automation (EDA) tools rely on benchmarks—designs written in Hardware Description Languages (HDLs) such as Verilog, SystemVerilog, or VHDL. Although collections of benchmarks for these languages exist, they are typically limited in size and variety. This scarcity has recently drawn more attention due to the increasing need for training large language models in this domain. To deal with such limitation, this paper presents a methodology and a corresponding tool for generating realistic Verilog designs. The tool, called ChiGen, was originally developed to test the Jasper<sup>®</sup> Formal Verification Platform, a product by Cadence Design Systems. Released as open-source software, ChiGen has been able to identify zero-day bugs in a range of tools, including Verible, Verilator, and Yosys. This paper outlines the principles behind ChiGen’s design, focusing on three aspects: (i) generation guided by probabilistic grammars, (ii) type inference using the Hindley-Milner algorithm, and (iii) code injection enabled by data-flow analysis. Once deployed on standard hardware, ChiGen outperforms existing Verilog Fuzzers such as Verismith, TransFuzz, and VlogHammer regarding structural diversity, code coverage, and bug-finding ability.

**Keywords:** Verilog, Synthesis, Testing, Fuzzing.

# List of Figures

|      |                                                                                                                                                                 |    |
|------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 3.1  | (a-b) Designs that uncovered issues in VERIBLE. (c) Design that uncovered issues in YoSys. . . . .                                                              | 28 |
| 3.2  | Overview of ChiGen’s modulus operandi. . . . .                                                                                                                  | 30 |
| 3.3  | Two probabilistic grammars. They recognize the same language, albeit with two different contexts of probabilities. . . . .                                      | 31 |
| 3.4  | Example of a module that performs a bitwise AND operation over two inputs. . . . .                                                                              | 32 |
| 3.5  | Snippet of the trace output from the design in figure 3.4. . . . .                                                                                              | 32 |
| 3.6  | List of reduction blocks. . . . .                                                                                                                               | 32 |
| 3.7  | Script to count the frequency of productions. . . . .                                                                                                           | 34 |
| 3.8  | Function to build the syntax tree from a grammar distribution. . . . .                                                                                          | 35 |
| 3.9  | Skeleton generation and symbol renaming. . . . .                                                                                                                | 36 |
| 3.10 | Generic node class from which all production classes inherit. It defines the common data members and methods. . . . .                                           | 37 |
| 3.11 | Identifier renaming through syntax tree traversal. . . . .                                                                                                      | 38 |
| 3.12 | Example of synthetic designs with instantiable namespaces. . . . .                                                                                              | 39 |
| 3.13 | Hindley-Milner Type Inference. . . . .                                                                                                                          | 40 |
| 3.14 | Default type inference visitor. . . . .                                                                                                                         | 40 |
| 3.15 | Unification algorithm. . . . .                                                                                                                                  | 43 |
| 3.16 | (a) Injection of function call. (b) Injection of hierarchical reference. . . . .                                                                                | 44 |
| 3.17 | The algorithm that implements module injection: it injects an instantiation of M in the program point p of C. . . . .                                           | 45 |
| 3.18 | Example of a call graph for module instantiation and hierarchical referencing. Nodes represent module names, while edge labels indicate instance names. . . . . | 47 |
| 3.19 | Example of a design after module injection. . . . .                                                                                                             | 48 |
| 3.20 | Example of an AST with the program points generated. . . . .                                                                                                    | 48 |
| 3.21 | SystemVerilog design with arithmetic expressions and vector assignments. . . . .                                                                                | 51 |
| 3.22 | SystemVerilog design with hierarchical reference and cross-module assignment. . . . .                                                                           | 51 |
| 3.23 | SystemVerilog design featuring non-ANSI port declarations and function scoping. . . . .                                                                         | 52 |
| 4.1  | Specification filtered out by syntactic verification. . . . .                                                                                                   | 55 |
| 4.2  | Verilog specification that fails the semantic test. . . . .                                                                                                     | 55 |

---

|      |                                                                                                                                                                                                         |    |
|------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 5.1  | Syntactical diversity of ChiGen designs, measured as the number of unique production rules in the Verilog grammar exercised when parsing a population of generated files. . . . .                       | 58 |
| 5.2  | Diversity of ChiGen designs, measured as the number of unique 4-grams: the number of different sequences formed by four kinds of tokens. . . . .                                                        | 59 |
| 5.3  | Branch coverage obtained by testing Verible's syntactic analyzer, formatter, Iverilog, and Verilator. . . . .                                                                                           | 60 |
| 5.4  | Issue 1225 in Icarus Verilog: The tool freezes when computing an invalid infinite loop. . . . .                                                                                                         | 61 |
| 5.5  | Issue 2181 in Verible's parser: The tool crashes instead of reporting syntax errors related to instantiation type. . . . .                                                                              | 62 |
| 5.6  | Issue 2359 in Verible's formatter: The tool fails to parse dash in front of unary operation (- -1). . . . .                                                                                             | 62 |
| 5.7  | Issue 1174 in Icarus Verilog: The tool crashes when assigning to parameters in a procedural block. . . . .                                                                                              | 62 |
| 5.8  | Time to generate 100 designs, setting the lower limit of tokens to 150. Numbers in boxes show the average number of tokens. . . . .                                                                     | 64 |
| 5.9  | The evolution of ChiGen, until its announcement as an open-source tool in October 2024. Each dot refers to 1,000 designs. Starting in v0.12, module injection uses a lower limit of 100 tokens. . . . . | 65 |
| 5.10 | Variation of the percentage of valid designs with the lower bound on the number of tokens in the designs produced by ChiGen. . . . .                                                                    | 66 |

# List of Tables

|     |                                                                                                                                                                                 |    |
|-----|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----|
| 5.1 | Issues discovered in various Verilog tools via tests using ChiGen designs. . . .                                                                                                | 61 |
| 5.2 | Crashes observed with 3,000 designs. ‘Asan/!Asan’: tool compiled with/without AddressSanitizer; CB: ChiBench; VS: Verismith; VH: VlogHammer; CG: ChiGen; TF: TransFuzz. . . . . | 63 |

# Contents

|          |                                                        |           |
|----------|--------------------------------------------------------|-----------|
| <b>1</b> | <b>Introduction</b>                                    | <b>14</b> |
| 1.1      | The Contributions of this Work . . . . .               | 14        |
| 1.1.1    | Software and Publications . . . . .                    | 16        |
| <b>2</b> | <b>Literature Review</b>                               | <b>17</b> |
| 2.1      | Testing of Programming Language Systems . . . . .      | 17        |
| 2.1.1    | Benchmarks . . . . .                                   | 17        |
| 2.1.2    | LLMs . . . . .                                         | 19        |
| 2.2      | Fuzzing . . . . .                                      | 22        |
| 2.2.1    | Fuzzing in General . . . . .                           | 22        |
| 2.2.2    | Fuzzing of Compilers . . . . .                         | 23        |
| 2.2.3    | Fuzzing of Verilog . . . . .                           | 25        |
| 2.3      | Final Remarks . . . . .                                | 27        |
| <b>3</b> | <b>Overview</b>                                        | <b>28</b> |
| 3.1      | Examples of Bugs that ChiGen can Discover . . . . .    | 28        |
| 3.2      | Bottom-Up Fuzzing . . . . .                            | 29        |
| 3.2.1    | Syntax Generation via Probabilistic Grammars . . . . . | 30        |
| 3.2.1.1  | Context-Sensitive Probabilities . . . . .              | 30        |
| 3.2.1.2  | Building a PCFG . . . . .                              | 31        |
| 3.2.2    | Variable Renaming and Scope Creation . . . . .         | 36        |
| 3.2.2.1  | Renaming Variables . . . . .                           | 36        |
| 3.2.2.2  | Dealing with Instantiable Namespaces . . . . .         | 38        |
| 3.2.3    | Type Inference via Unification . . . . .               | 39        |
| 3.2.3.1  | Constraint Generation . . . . .                        | 40        |
| 3.2.3.2  | Unification . . . . .                                  | 42        |
| 3.2.4    | Code Expansion via Code Injection . . . . .            | 44        |
| 3.2.4.1  | Reaching Definition Analysis . . . . .                 | 46        |
| 3.3      | Using ChiGen . . . . .                                 | 49        |
| 3.3.1    | Training a Grammar . . . . .                           | 49        |
| 3.3.2    | Generating Designs . . . . .                           | 50        |
| 3.4      | Design Examples . . . . .                              | 50        |
| <b>4</b> | <b>ChiBench: The Training Set</b>                      | <b>54</b> |

|          |                                    |           |
|----------|------------------------------------|-----------|
| 4.1      | Curating the Data . . . . .        | 54        |
| <b>5</b> | <b>Evaluation</b>                  | <b>57</b> |
| 5.1      | RQ1 – Diversity . . . . .          | 57        |
| 5.2      | RQ2 – Coverage . . . . .           | 59        |
| 5.3      | RQ3 – Bugs . . . . .               | 61        |
| 5.4      | RQ4 – Latency . . . . .            | 63        |
| 5.5      | RQ5 – Software Evolution . . . . . | 64        |
| <b>6</b> | <b>Conclusion</b>                  | <b>67</b> |
| 6.1      | Limitations . . . . .              | 67        |
| 6.2      | Future Work . . . . .              | 68        |
|          | <b>Bibliography</b>                | <b>70</b> |

# Chapter 1

## Introduction

Fuzzing [Takanen et al., 2018] is an automated testing technique that generates random—often unexpected—inputs to a program to discover bugs, vulnerabilities, or unexpected behaviors. Many EDA (Electronic Design Automation) tools such as YOSYS, VERILATOR, MODELSIM<sup>®</sup>, XCELIUM<sup>™</sup>, JASPER<sup>®</sup> and, Design Compiler<sup>®</sup> can benefit from a fuzzer that generates Verilog designs automatically. This benefit comes in the form of early bug discovery, performance optimizations, compliance checking, and general validation.

There exist open-source Verilog fuzzers, such as VlogHammer [Wolf, 2021], Verismith [Herklotz and Wickerson, 2020], and TransFuzz [Solt and Razavi, 2025]. These tools operate in a top-down fashion: starting with a minimal core of valid Verilog syntax and expanding it through various techniques, always ensuring the generation of semantically valid designs. However, our experience using these tools to test the Jasper Formal Verification Platform suggests that the requirement to produce only valid Verilog designs limits the diversity of their test cases. As outlined in Chapter 3, semantically invalid Verilog specifications can be equally effective as valid ones in uncovering crashes in EDA tools. Additionally, these tools often produce syntactic constructs that differ from those found in human-written code, which is not ideal, since fuzzers are expected to simulate real use, and not only random testcases. Furthermore, as detailed in Chapter 5.1, these tools cover fewer than 40% of the production rules in a grammar of Verilog-2005 (IEEE 1364-2005) by Association et al. [2006].

### 1.1 The Contributions of this Work

This work presents ChiGen, a “bottom-up” fuzzer designed to test the Jasper Formal Verification Platform from Cadence Design Systems. In October 2024, ChiGen was released as an open-source tool. Since then, it has received contributions from academics and engineers, evolving into an effective Verilog fuzzer. Unlike state-of-the-art Verilog

fuzzers, ChiGen generates designs in a bottom-up fashion. It first produces a syntactically valid design with placeholders for user-defined symbols, such as variables, modules, and functions. These placeholders are then replaced through multiple inference steps, transforming the skeleton into a valid Verilog design.

As described in Section 3.2, ChiGen operates in four stages: First, it generates the skeleton of a Verilog specification using a probabilistic grammar. The probabilities of production rules were trained on the benchmark suite detailed in Chapter 4. Second, ChiGen replaces mock identifiers with names that respect the scoping rules. Third, it applies the Hindley-Milner type inference algorithm [Hindley, 1969; Milner, 1978; Sulzmann, 2000], commonly used in functional programming, to infer the types of variables. Finally, it employs a technique recently proposed by Li et al. [2024] to combine Verilog modules, functions, and references, achieving any predefined number of tokens.

As discussed in Chapter 5, ChiGen surpasses top-down fuzzers such as Verismith, VlogHammer, and TransFuzzer in code coverage, structural diversity, and bug-finding effectiveness. Since its release, it has uncovered issues, such as those described in Chapter 3, in open-source tools such as Yosys, Verilator, Icarus, and Verible. These successes stem from the following contributions:

- **Real-World Training Set:** ChiGen emulates the syntax of real Verilog designs. To train it, we curated a dataset of 50,000 designs mined from open-source repositories with permissive licenses. This benchmark suite, known as ChiBench, is a contribution in itself and has been used for tasks beyond ChiGen training, as discussed in Chapter 4.
- **Probabilistic Grammar:** As detailed in Subsection 3.2.1, ChiGen can be trained on any number of Verilog examples without human intervention. The more designs it observes during training, the more realistic the designs that it generates.
- **Inference Mechanism:** A trained instance of ChiGen produces syntactically valid Verilog skeletons, which are then refined through static analyses. These analyses ensure that variables are defined before use (Subsection 3.2.2) and that all references adhere to declared types (Subsection 3.2.3).
- **Code Injection:** ChiGen incrementally integrates generated components into more complex designs, linking modules via bindings, instantiations, function calls, or hierarchical references (Subsection 3.2.4). As new modules, functions, and references are generated, they become available for insertion into the ongoing design.

### 1.1.1 Software and Publications

The main product of this work is the ChiGen fuzzer and the ChiBench suite of Verilog designs. Both tools are publicly available under the GPL 3.0 license and can be retrieved at <https://github.com/lac-dcc/chimera>. Additionally, this work inspired the publication of two papers, the first of which is under submission at the time this dissertation is being prepared:

1. João Victor Amorim, Luiza Melo, Rafael Sumitani, Raissa Maciel, Augusto Mafra, Mirlaine Crepalde, Fernando Magno Quintão Pereira: *Bottom-Up Generation of Verilog Designs for Testing EDA Tools*. arXiv:2504.06295v1 [cs.AR] (2025) Sumitani et al. [2024].
2. Rafael Sumitani, João Victor Amorim, Augusto Mafra, Mirlaine Crepalde, Fernando Magno Quintão Pereira: *ChiBench: a Benchmark Suite for Testing Electronic Design Automation Tools*. CoRR abs/2406.06550 (2024) Vieira et al. [2025].

**A Work of Many Hands.** ChiGen is the product of a collaborative effort by Cadence Design Systems and the Federal University of Minas Gerais (UFMG). Since its public release, the tool has grown through the input and feedback of multiple programmers. Notably, external developers such as Rishik Ram and Mohit Karekar have made significant code contributions, helping to expand and refine its capabilities.

Although ChiGen’s development has benefited from this cooperation, its core conception, architectural design, and initial implementation are my own. One of the fundamental design decisions was to adopt a bottom-up fuzzing strategy, first generating the structural skeleton of a Verilog design, and then filling it up with actual code. This methodology has proven effective in producing syntactically valid and structurally varied test cases suitable for evaluating EDA tools.

# Chapter 2

## Literature Review

This chapter presents a brief overview of the literature related to fuzzing, going from the testing of generic systems towards the more specific problem of fuzzing Verilog designs. Thus, the goal of the chapter is to give the readers enough data to position this dissertation concerning the vast amount of academic work that came out related to Testing of Programming Language Systems (Section 2.1) and Fuzzing (Section 2.2).

### 2.1 Testing of Programming Language Systems

The testing of programming language systems consists of evaluating their correctness, reliability, and performance. A common approach involves using system-specific benchmarks to assess these aspects. Recently, however, the rise of Large Language Models (LLMs) for code generation has introduced new challenges and opportunities for testing such systems.

This chapter is organized into two parts:

- Benchmarks (Subsection 2.1.1)
- LLMs (Subsection 2.1.2)

#### 2.1.1 Benchmarks

ChiGen can be used to generate Verilog benchmarks; however, it is not a static benchmark collection. Numerous established collections already exist, composed of hardware designs written in specification languages such as Verilog and VHDL [Amaru et al., 2019; Babb et al., 1997; Brglez et al., 1989; Koźmiński, 1991; Murray et al., 2015; Sumitani

et al., 2024]. These repositories have long supported the evaluation of synthesis tools, the validation of design methodologies, and comparative studies in hardware research.

Lu et al. [2023] introduce RTLLM, an open-source benchmark designed to evaluate the capabilities of large language models in generating Register-Transfer Level (RTL) designs from natural language instructions. RTLLM addresses limitations in previous work, where evaluation often focused solely on functional correctness and was restricted to small designs curated by the author, making fair comparisons difficult. To provide a more accurate estimate, the benchmark evaluates generated designs across three dimensions: syntax correctness, functional correctness, and design quality. In addition, the authors propose a simple but effective prompt-engineering strategy called self-planning, which significantly improves the performance of GPT-3.5 on this benchmark.

Liu et al. [2023] introduce VerilogEval, a benchmarking framework designed to evaluate LLMs on Verilog code generation tasks. The benchmark includes 156 diverse problems sourced from HDLBits, ranging from basic combinational logic to complex finite-state machines. The framework supports automated functional verification through simulation-based comparison with reference outputs. The authors demonstrate that LLM performance can be improved through supervised fine-tuning using synthetically generated problem-code pairs.

Guo and Zhao [2025] present ResBench, a benchmark designed to evaluate LLM-generated HDL code for FPGAs, focusing on hardware resource efficiency, rather than functional correctness. Traditional benchmarks have largely overlooked the resource consumption characteristics of HDL code, although efficient use of resources such as lookup tables (LUTs) is important for real-world FPGA applications. ResBench offers a diverse set of 56 problems from 12 distinct categories, covering designs from finite-state machines to financial computing. In addition to the benchmark, the authors provide an automated evaluation framework that verifies functional correctness and quantifies the usage of hardware resources, enabling systematic comparison of LLMs on both fronts. Their experiments show a significant variance in the quality and resource efficiency of the code generated by different models.

In contrast to ChiGen, these traditional collections are immutable: their content is manually created or mined from existing repositories and, once published, can only be updated through deliberate manual effort. This immutability limits their adaptability to newly emerging synthesis features and growing toolchains.

While these benchmarks are static, ChiGen offers a dynamic generation of benchmarks. Each execution produces a new and different design, which makes the dataset extensible by default. Furthermore, specific benchmark instances can be seeded or incorporated into ChiGen’s internal dataset, allowing users to guide generation towards particular structures or testing goals. This flexibility makes ChiGen a powerful complement to static benchmark suites, specifically in contexts that demand diversity, scalability,

or targeted stress testing.

### 2.1.2 LLMs

Over the last two years, the LLM revolution has brought to light many language models for Verilog. Chang et al. [2023] introduce ChipGPT, a framework designed to assess how effectively large language models can automate hardware logic design from natural language specifications. The authors propose a scalable four-stage, zero-code development flow that leverages LLMs without the need for fine-tuning or retraining. In this workflow, ChipGPT first generates LLM prompts based on a user-provided description, and then uses the model's output to produce initial Verilog code. An output manager subsequently refines and optimizes the generated HDL, assembling a candidate design space. Finally, ChipGPT performs a targeted search across this space to identify the implementation that best meets the specified hardware metrics. Their results demonstrate that ChipGPT enhances both programmability and controllability when compared to baseline LLM outputs, while also offering a broader and more optimized design space.

Thakur et al. [2022] investigate the capacity of large language models to automate Verilog RTL code generation by benchmarking both pre-trained and fine-tuned models on a dedicated Verilog dataset. Their work highlights the potential of LLMs to assist hardware designers by reducing human error and improving productivity in digital system design. They fine-tune open-source LLMs, such as CodeGen, on datasets sourced from GitHub repositories and Verilog textbooks, and introduce an evaluation pipeline that tests both syntactic validity and functional correctness through automated test benches. Their findings show that fine-tuning enhances the syntactic correctness of generated code, boosting it to 25.9% across different design scenarios. The fine-tuned open-source CodeGen model surpasses OpenAI's commercial Codex model in functional correctness by 6.5%.

Thakur et al. [2024a] propose VeriGen, a large language model fine-tuned specifically for Verilog code generation, addressing the growing need for automation in hardware design. By training an open-source model, CodeGen-16B, on datasets curated from GitHub repositories and Verilog textbooks, the authors demonstrate that domain-specific fine-tuning significantly improves syntactic and functional correctness. Their model outperforms GPT-3.5-turbo by 1.1% in overall functional accuracy and achieves a 41% increase in syntactic correctness compared to the base model.

Gao et al. [2024] present AutoVCoder, an open-source framework that takes advantage of LLMs focused on improving the correctness and quality of automatically-generated Verilog designs. By collecting a high-quality hardware designs dataset, along with a

retrieval-augmented generation (RAG) mechanism, the authors were able to fine-tune an LLM and achieve a 0.5% and 2.2% improvement in functional correctness on the EvalMachine and EvalHuman benchmarks compared with BetterV, and a 3.4% increase in both syntax correctness and functional correctness on the RTLMLM benchmark compared to RTLCoder.

Wang et al. [2024] present an approach to train LLMs to generate Verilog code using reinforcement learning. The suggested reward function is based on the similarity between the Abstract Syntax Tree (AST) of the design generated by the language model and the golden standard, followed by human feedback, penalizing invalid code, and rewarding syntactically correct code. The method achieves state-of-the-art results, outperforming models with 13 and 16 billion parameters while having only 6.7 billion parameters.

Driven by the lack of high-quality instruction-tuning data for Verilog generation, Zhao et al. [2024] introduces CodeV, a suite of Verilog-generation LLMs. Observing that LLMs tend to produce more accurate natural language summaries of HDL code than they do generating Verilog itself, the authors propose taking advantage of this capability for data generation. By prompting LLMs with Verilog code and collecting the resulting natural language descriptions, they construct a high-quality dataset for instruction tuning. Models fine-tuned on this dataset show promising results, surpassing previous open-source state-of-the-art models such as BetterV and RTLCoder, as well as proprietary systems like GPT-4.

Thakur et al. [2024b] propose AutoChip, a framework that enhances HDL code generation using feedback loops between large language models and HDL simulation tools. Instead of relying solely on static text-to-code generation, AutoChip introduces a dynamic process where LLMs iteratively refine their output based on real-time error messages and simulation feedback, similar to how human designers debug and revise code. Using tools like Icarus Verilog for compilation and simulation, AutoChip guides LLMs to correct syntactic and semantic errors during generation, improving the functional correctness of the generated Verilog modules. The experiments, carried out on the HDLBits problem set, show that this feedback-driven approach can increase the accuracy of the code, achieving a gain of 24.2% over traditional generation methods.

Tang et al. [2024] present HiVeGen, a hierarchical LLM-based framework designed to improve Verilog code generation for scalable hardware designs. Apart from conventional LLM-based approaches, which typically generate flat and isolated HDL blocks, HiVeGen mirrors handcrafted chip design practices by decomposing complex systems into hierarchical submodules that are more manageable for LLMs. This hierarchical strategy reduces the likelihood of hallucination – mainly in the context of domain-specific accelerators (DSAs) – and also combines automated Design Space Exploration (DSE) into the generation process. Additionally, HiVeGen introduces a weight-based retrieval mechanism to enhance code reuse and facilitate hierarchy-aware prompt generation. The

framework also supports real-time human-in-the-loop interactions, which help minimize error correction overhead and improve the overall quality of the generated Verilog designs.

Sun et al. [2025] address LLM hallucinations in HDL code generation by LLMs through a structured, expert-inspired methodology. Rather than relying on single-pass generation, the authors introduce paradigm blocks, modular templates that mirror the typical workflows of human hardware designers, including information extraction, design flow decomposition, and external tool integration. Their approach guides LLMs to classify the target circuit type and apply the corresponding paradigm block to generate Verilog code more systematically. In addition, they propose a two-phase iterative generation process designed to improve the pass rate of automatically generated testbenches within a constrained number of iterations. Experimental results show that this method improves the functional correctness of HDL outputs compared to baseline LLM prompting.

Liu et al. [2025] present DeepRTL, a unified representation model designed to handle both Verilog understanding and generation tasks. While most prior research focuses exclusively on generating Verilog from natural language prompts, DeepRTL focuses on understanding HDL code. Built on the CodeT5+ backbone, the model is fine-tuned on a rich, multi-level dataset that aligns Verilog designs with natural language descriptions, designed to improve the semantic alignment between code and text. In addition to proposing this model, the authors introduce the first dedicated benchmark for Verilog understanding and apply evaluation metrics such as embedding similarity and GPT Score – moving beyond the traditional metrics BLEU and ROUGE. Their results demonstrate that DeepRTL surpasses GPT-4 in understanding tasks and achieves a generation performance comparable to OpenAI’s o1-preview model.

These works illustrate the effectiveness of tailoring LLMs to domain-specific programming languages and support the broader trend toward applying LLMs beyond general-purpose software engineering tasks. It is particularly relevant for research on AI-assisted electronic design automation and domain-specialized LLMs. However, LLMs show a limitation concerning the effort, time, and computational power to train and maintain one. ChiGen, on the other hand, is not a language model; it is a fuzzer, meaning it does not attempt to shape the code towards any specific semantics. ChiGen does use a probabilistic grammar that models the probability of production rules as k-grams; however, it does not assign probabilities to sequences of tokens – rather, it assigns them to sequences of production rules. It is much more cost-effective when it comes to producing random Verilog testcases.

## 2.2 Fuzzing

Another branch of system testing is fuzzing. Fuzzing is a powerful testing technique, mostly used to uncover software bugs and vulnerabilities. Its core methodology involves generating and injecting a target system with valid, invalid, and malformed inputs to provoke unexpected behavior and expose defects [Liang et al., 2018]. Due to its effectiveness, fuzzing has broad applicability across diverse domains, from security testing to hardware verification. To introduce and explain this concept, this section explores it in a three-part survey:

- Fuzzing in General (Subsection 2.2.1)
- Fuzzing of Compilers (Subsection 2.2.2)
- Fuzzing of Verilog (Subsection 2.2.3)

### 2.2.1 Fuzzing in General

Yun et al. [2022] survey embedded fuzzing techniques, emphasizing their relevance in identifying and mitigating vulnerabilities in tools' firmware. For example, drones, which are used in the public and private sectors, face security challenges such as GPS spoofing, implementation errors, and software vulnerabilities. Although countermeasures for GPS spoofing have been proposed, less attention has been paid to addressing software errors in these devices, which could lead to serious implications if left unaddressed. Although the development of fuzzers specifically built for embedded systems is behind that of traditional systems, multiple embedded-specific fuzzing tools have appeared. This work provides a detailed analysis of these tools, focusing on their strengths, limitations, and applications to address the growing security needs in embedded systems.

Xia et al. [2023] introduce Fuzz4All, a universal fuzzing framework based on Large Language Models to generate and mutate test inputs across a wide variety of programming and formal languages. Fuzz4All is designed to be language-agnostic and capable of exploring diverse syntactic and semantic features of any target language. This is achieved through an autoprompting technique that dynamically crafts LLM prompts for input generation and guides the LLM-powered fuzzing loop to iteratively refine these inputs. The authors evaluate Fuzz4All in nine systems that process six different languages – including C, C++, Go, SMT2, Java, and Python – and demonstrate that the tool consistently

outperforms language-specific fuzzers in terms of code coverage. Fuzz4All uncovered 98 previously unknown bugs in software, such as GCC, Clang, Z3, CVC5, OpenJDK, and Qiskit, 64 of which were confirmed by developers.

Daniele et al. [2023] focus on the challenges posed by fuzzing stateful systems, which maintain internal states that complicate the fuzzing process. They present the first class of fuzzers specially designed for those systems, providing a detailed classification and comparison of the existing strategies. The paper offers an analysis of the approaches developed to address these challenges, categorizing fuzzers based on their methods and effectiveness. By providing this framework, the authors help clarify the variety of strategies available for fuzzing stateful applications, thus contributing to understanding how to approach this complex problem in security testing.

Huang et al. [2024] present a survey on the intersection of fuzz testing and LLMs. The paper reviews state-of-the-art approaches that combine LLMs with fuzzing, offering insights into their potential for fully automated vulnerability discovery. The authors execute a statistical analysis of the literature up to 2024 and discuss future directions for LLM-based fuzzing techniques, suggesting that LLMs will likely play a significant role in modern software testing workflows.

Zhang et al. [2025] provide a survey on the field of protocol fuzzing, addressing the growing concerns about vulnerabilities in protocol implementations. Their study offers an overview of the challenges involved in fuzzing communication protocols and categorizes the various research efforts dedicated to solving these issues. The authors analyze the existing literature and classify fuzzing techniques, exploring how different approaches tackle the complexities of protocol testing. In addition, the paper discusses potential future research directions, offering guidance for academic and industry practitioners.

## 2.2.2 Fuzzing of Compilers

Fuzzing compilers presents a particular set of challenges due to the complexity and specificity of compiler implementations. Unlike general-purpose programs, compilers are tied to the syntax and semantics of the languages they support, the intermediate representations they generate, and the optimization pipelines they employ. Each compiler may interpret edge cases differently, apply unique transformations, or target specific architectures, making generic fuzzing strategies less effective. As a result, effective compiler fuzzing requires specifically crafted tools that understand the complexities of both the source language and the internal logic of the compiler under test. These fuzzers must be built not only to generate syntactically valid inputs but also to exercise semantic paths,

trigger optimization routines, and expose subtle correctness or crash-inducing bugs that arise from the compiler’s internal state.

Csmith [Yang et al., 2011] is one of the most established tools for testing C compilers through randomized program generation. It produces syntactically correct C programs that avoid undefined and unspecified behaviors. To detect compiler errors, each generated program is compiled and executed by three different compilers; if one produces an output that diverges from the majority, the discrepancy signals a potential bug. This approach relies on the guarantee that every program has a single, well-defined sequence of side effects, referred to as the program’s meaning. Over three years, Csmith proved to be highly effective, reporting more than 300 bugs across various compiler implementations.

Livinskii et al. [2020] introduce an approach to test case reduction and deduplication in the context of compiler testing. Their method leverages transformation-based compiler testing, where test cases are systematically transformed to uncover compiler bugs. Innovation lies in integrating reduction and deduplication directly into the transformation process, thereby minimizing the overhead typically associated with these tasks. This integration allows for efficient identification and elimination of redundant test cases, thus streamlining the testing pipeline. The authors demonstrate the effectiveness of their approach through extensive experiments, showing significant improvements in both the speed and quality of compiler testing.

Building on the limitations of traditional black- and gray-box fuzzers, Yang et al. [2024] introduce WhiteFox, a white-box compiler fuzzing framework powered by large language models. WhiteFox uses LLMs to generate test programs and to analyze compiler source code, allowing it to target logic bugs in compiler optimization stages. Applied to deep learning compilers like PyTorch Inductor and TensorFlow-XLA, WhiteFox discovered 92 previously unknown bugs and surpassed existing fuzzers by up to  $8\times$  in bug-finding efficiency.

Similarly, focusing on the compiler’s infrastructure internal layers, Rong et al. [2024] propose IRFuzzer, a fuzzing framework aimed directly at the LLVM backend. Instead of working with high-level code, IRFuzzer operates on LLVM Intermediate Representation (IR) and uses constrained mutations to ensure valid program generation. It also introduces matcher table coverage, a feedback mechanism that helps guide test case generation by monitoring backend execution paths. Across 29 LLVM backend targets, IRFuzzer reported 74 new bugs, 49 of which have already been fixed.

Elnaggar et al. [2025] tackle cross-architecture compiler testing by combining Large Language Models with Retrieval-Augmented Generation (RAG). Their approach automates the creation of architecture-diverse test cases, reducing manual effort for developers. Applied to the Intel DPC++/C++ compiler, the method uncovered 87 issues of output mismatches and runtime errors across various hardware platforms within hours.

Lastly, Suo et al. [2025] introduce DESIL, a fuzzing framework designed to detect

silent bugs—logic errors that do not cause crashes but lead to incorrect program behavior—in MLIR-based compiler infrastructures. DESIL eliminates undefined behaviors from input programs and optimizes the selection of lowering paths for efficient execution and comparison. During this process, the framework identified 23 silent bugs and 19 crash bugs, with most issues confirmed or resolved.

### 2.2.3 Fuzzing of Verilog

Several tools share similar goals applied to the HDL domain. One such tool is Verismith [Herklotz and Wickerson, 2020], which aims to uncover bugs by generating random behavioral code that avoids undefined values. Verismith constructs a design incrementally—line by line—while maintaining awareness of the current context to ensure code validity. Once a complete design is generated, it is passed to an EDA tool, which is expected to produce a corresponding netlist, as all generated designs are valid by construction. The tool then checks for functional equivalence between the original Verilog design and the generated netlist. Any mismatch or crash during synthesis indicates the presence of a bug. Using this methodology and generating a synthesizable subset of Verilog 2005, Verismith successfully identified eleven bugs across multiple tools, including Yosys, Vivado, XST, and Quartus Prime.

LegoHDL [Xu et al., 2024] introduces an AST-guided HDL code generator that takes advantage of a library of functional blocks derived from cyberphysical systems. This enables the creation of more complex, diverse, and defect-triggering designs across Verilog, VHDL, and SystemVerilog. LegoHDL applies differential testing across multiple tools and has discovered 20 new synthesis-related bugs, demonstrating superior coverage and bug-finding capacity compared to previous tools.

TransFuzz [Solt and Razavi, 2025] generates randomized RTL designs to uncover translation vulnerabilities in EDA tools. These flaws may allow attacker-crafted hardware Trojans – called MIRTL gadgets – to pass undetected through conventional verification workflows and be synthesized into real hardware. These gadgets exploit weaknesses in how EDA tools translate RTL to lower-level representations, allowing confused deputy attacks. TransFuzz explores this space by generating complex RTL designs with rich operator interconnections and then compares the outputs from multiple RTL simulators to detect inconsistencies. Using this approach, TransFuzz discovered 20 translation vulnerabilities among 31 previously unknown bugs (including 25 CVEs) in four widely used open-source EDA tools. The authors demonstrate how MIRTL gadgets can significantly strengthen traditional backdoors against white-box defenses, including a real-world ex-

ample integrated into the CVA6 RISC-V core.

Canakci et al. [2022] introduced ProcessorFuzz, a fuzzing framework designed for processor verification at the Register-Transfer Level. The authors address the limitations of prior hardware fuzzers, such as limited support for mainstream Hardware Description Languages and the reliance on misleading coverage signals that fail to properly guide input generation towards unexplored and significant processor states. To overcome these challenges, ProcessorFuzz introduces a coverage metric based on Control and Status Register (CSR) transitions. Since CSRs reflect and influence the internal state of a processor, tracking their state changes allows the fuzzer to guide its exploration of the design space. ProcessorFuzz is HDL-agnostic and does not require invasive instrumentation of the design under test, making it applicable to a wide range of processors. The tool was evaluated on three open-source RISC-V cores – Rocket, BOOM, and BlackParrot— where it outperformed existing solutions such as DIFUZZRTL in bug discovery speed and coverage. The authors validated the tool’s effectiveness by uncovering nine previously unknown bugs, all of which were later confirmed by the respective hardware maintainers.

Similarly, Gadde et al. [2024] address the challenge of improving coverage and bug discovery in RTL designs through FuzzWiz, an automated hardware fuzzing framework. The authors propose a methodology that closes the gap between software fuzzing techniques and hardware verification by converting RTL modules into C/C++ models. Once transformed, FuzzWiz generates generic testbenches equipped with assertions, and prepares the design for fuzzing through fuzzer-specific compilation, linking, and execution. This workflow is fully automated and configurable, including debugging support for cases where a crash is detected during fuzzing. The authors validate FuzzWiz using four intellectual property (IP) blocks from Google’s OpenTitan chip, demonstrating that the framework achieves around 90% coverage ten times faster than traditional simulation-based regression techniques.

Zhou et al. [2025] presents *VeriXmith*, a framework for cross-checking Verilog compilers by leveraging semantic equivalence across different circuit representations. Their approach interconnects synthesis and simulation tools—such as those generating Verilog netlists or C++ simulation code—to detect inconsistencies via canonical representations of circuit behavior. The authors introduce mutation operators that expand Verilog test inputs by injecting diverse data/control paths and syntactic constructs, thereby increasing the likelihood of exposing latent toolchain bugs. This methodology led to the discovery of 31 previously unknown issues in widely used open-source tools like Verilator and Yosys, demonstrating the effectiveness of semantic-aware differential testing in uncovering compiler defects.

## 2.3 Final Remarks

While LLM-based approaches have significantly advanced the state of HDL code generation, and while high-quality benchmarks continue to improve evaluation practices, a crucial gap remains: the ability to generate diverse, scalable, and easily customizable test cases without the overhead of fine-tuning large models or curating static datasets. Static benchmarks, although indispensable for standardized evaluation, are inherently rigid and slow to adapt to evolving toolchains and synthesis features. LLMs, on the other hand, introduce considerable demands in terms of computational cost, data requirements, and prompt engineering effort. These limitations underscore the need for a complementary solution—one that emphasizes flexibility, extensibility, and efficiency. This motivates the present work, which introduces ChiGen: a lightweight, probabilistic grammar-based fuzzer capable of generating structurally rich and semantically diverse Verilog designs for dynamic testing scenarios.

The techniques proposed here focus specifically on the construction of Verilog fuzzers. Although several open-source Verilog fuzzers have been developed [Herklotz and Wickerson, 2020; Solt and Razavi, 2025; Wolf, 2021], they typically rely on incremental expansion strategies—starting from a core syntactic seed and growing valid designs through localized decisions. In contrast, ChiGen adopts a more global, modular generation process. During its development, we had the opportunity to engage with the authors of Verismith, and a comment shared in personal communication encapsulates the distinction: “I think the main difference between the two approaches [...] is that Verismith generates a set of Verilog modules one line at a time, making decisions locally about what statement to generate next. We also perform the entire generation in a single step, so splitting up different phases into separate steps is very interesting. Again, this could be seen as a local approach versus a more global and modular approach in ChiGen.” [Herklotz, 2024]

# Chapter 3

## Overview

This chapter explains the design and implementation of ChiGen. However, before diving into its minutiae, Section 3.1 presents examples of bugs it can uncover in EDA tools, as a way to motivate the decisions introduced in Section 3.2.

### 3.1 Examples of Bugs that ChiGen can Discover

This section illustrates the usage of ChiGen with the three designs seen in Figure 3.1. These designs were automatically produced by ChiGen, using a probabilistic grammar with a context of length one (the notion of context depth shall be explained in Subsection 3.2.1).

|                                                                   |                                                                                                                    |                                                                                                 |
|-------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------|
| <p>(a)</p> <pre><b>module</b> module_0(); <b>endprogram</b></pre> | <p>(b)</p> <pre><b>module</b> module_0(id_1);   <b>output logic</b>     <b>signed</b> id_1; <b>endmodule</b></pre> | <p>(c)</p> <pre><b>localparam</b> id_1 = id_1; <b>module</b> module_0(); <b>endmodule</b></pre> |
|-------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------|

Figure 3.1: (a-b) Designs that uncovered issues in VERIBLE. (c) Design that uncovered issues in YOSYS.

All these designs have revealed an issue in some EDA tool. All the issues were reported and acknowledged. Throughout this work, we shall say that a Verilog design is valid if it passes Jasper’s analysis phase (e.g., it successfully goes through the **analyze** command). Thus, invalid designs either show incorrect syntax or fail the static semantic analysis.

**Tool Accepts Invalid Syntax:** The first code, in Figure 3.1 (a), uncovered a bug in Verible’s parser. The keyword **endprogram** incorrectly matches the keyword **endmodule**. However, even though the design is syntactically invalid, it was accepted by Verible’s parser. In this case, the expected behavior would be to report the syntactic error.

**Tool Crashes on Valid Syntax:** The code in Figure 3.1 (b) is valid; however, the port `id_1` is declared as `output logic signed`. Although valid, this syntax is not typical in standard SystemVerilog, since usually ports have no signal. Nevertheless, this specification causes a segmentation fault in Verible’s parser.

**Tool Crashes on Invalid Semantics:** The design in Figure 3.1 (c) is syntactically valid. The syntax for declaring a `localparam` outside any module is acceptable. However, the code is not semantically valid. The issue lies in the `localparam` declaration: the line `localparam id_1 = id_1;` attempts to assign the value of `id_1` to itself, which creates a circular reference. In Verilog, `localparam` must be initialized to a constant expression or a value known at compile time; hence, referring to itself in this way does not provide a valid initialization value. Since Yosys considers it a valid design, this module causes infinite recursion in Yosys when it tries to optimize the generated AST, eventually forcing a crash, as the tool runs out of stack memory.

In addition to the three examples described in this chapter, designs generated by ChiGen have discovered several other issues in popular EDA tools, including Icarus, Verilator, Verible’s obfuscator, and Verible’s formatter. Some of these issues have led to non-trivial changes in these tools. As an example, a ChiGen design provoked the addition of syntactic rules in Verible’s parser to mix anonymous and named instances. Similarly, at least two interventions were recently added to Verilator (Release 5.030, 2024-10-27) due to issues raised via ChiGen. ChiGen has also been successful in discovering issues in proprietary tools. However, in this work, we will only discuss issues that have been publicly reported.

## 3.2 Bottom-Up Fuzzing

ChiGen works in four phases. Figure 3.2 shows how these phases are related. The rest of this section describes each of the steps:

- Syntax Generation (Subsection 3.2.1).
- Variable Renaming and Scope Creation (Subsection 3.2.2);
- Type Inference (Subsection 3.2.3).
- Code Expansion (Subsection 3.2.4).

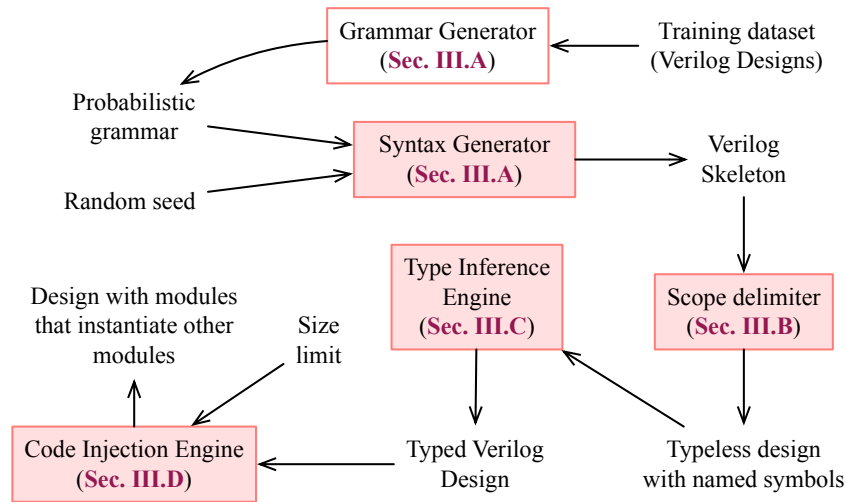


Figure 3.2: Overview of ChiGen’s modus operandi.

### 3.2.1 Syntax Generation via Probabilistic Grammars

To produce a Verilog specification, ChiGen begins by generating a “skeleton” of the design: a structure that adheres to Verilog’s syntactic rules. This skeleton is created using a Probabilistic Context-Free Grammar (PCFG), which assigns probabilities to sequences of production rules. These production rules were taken from Verible’s grammar, which parses the IEEE 1800-2017 standard. As seen in Figure 3.2, ChiGen is distributed with a grammar generator. This tool receives a training set (a collection of Verilog designs) as input. It parses every design in the training set, recording the number of times each grammar production was activated during parsing as a YAML file. The grammar generator uses this log to build the probabilities associated with each production rule in the Verilog grammar. The public distribution of ChiGen was trained with a collection of designs extracted from a benchmark suite called ChiBench. The construction of ChiBench is the subject of Chapter 4.

#### 3.2.1.1 Context-Sensitive Probabilities

In a traditional PCFG, each production rule’s probability is independent of the others, making it “memoryless” (or Markovian). Thus, the probability of applying a rule to a non-terminal depends only on the non-terminal itself, not on preceding or succeeding rules. However, context-sensitive probabilistic models allow for conditional dependencies

across rule applications, where the probability of a rule can depend on previously chosen rules, leading to “sequence-aware” probabilities. ChiGen enables conditional dependencies between rule applications, allowing the probability of a given rule to depend on previously selected  $N$  rules (an  $N$ -gram), resulting in “sequence-aware” probabilities. We limit the probability context  $N$  – the chain of production rules associated with a probability – to six productions, as each additional context introduces a potentially exponential increase in the table of probabilities. To construct the PCFG, ChiGen parses a training set of Verilog designs. It parses each file in this set, recording how often each sequence of productions is used during parsing. Example 1 shows instances of probabilistic grammars.

**Example 1** *Figure 3.3 shows two examples of PCFGs. The example in Figure 3.3 (a) does not consider context. The example in Figure 3.3 (b) considers contexts of depth one; that is, it can “remember” the rule that led to the production of the current nonterminal that must be expanded. As an illustration, the chance of adding a new element to a list of declarations decreases if we know that this list already has one element, as very long chains of declarations are uncommon.*

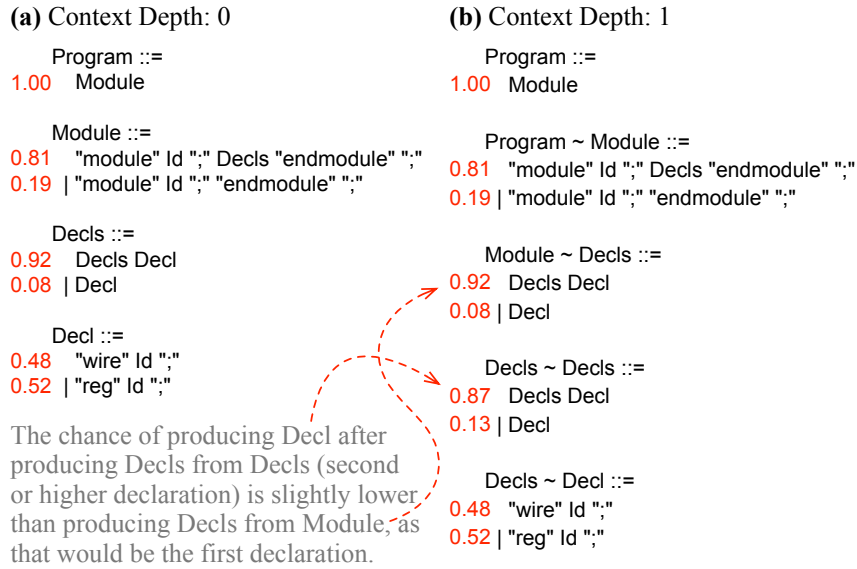


Figure 3.3: Two probabilistic grammars. They recognize the same language, albeit with two different contexts of probabilities.

### 3.2.1.2 Building a PCFG

To construct a PCFG, the first step is to collect design examples. As discussed in Chapter 4, we used the ChiBench dataset. The next step involves extracting the sequence

of grammar productions used during parsing. For this, we employed the Verible syntactic analyzer to generate a trace file, which logs all parser states and reductions that occur during analysis. The trace file is a debug option offered by parsers built using Bison.

```
1 module and2bit (input wire [1:0] a, input wire [1:0] b, output wire [1:0] y);
2     assign y = a & b;
3 endmodule
```

Figure 3.4: Example of a module that performs a bitwise AND operation over two inputs.

Figure 3.4 illustrates a simple bitwise-and design, from which a trace file can be generated. However, even for relatively small designs, these trace files can be extensive – ours typically exceed a thousand lines. A snippet of such a trace is shown in Figure 3.5.

```
Starting parse
Entering state 0
Reading a token: Next token is token "module" ()
Shifting token "module" ()
Entering state 36
Reducing stack by rule 1224 (line 5457):
    $1 = token "module" ()
-> $$ = nterm module_start ()
Stack now 0
Entering state 123
Reading a token: Next token is token SymbolIdentifier ()
Reducing stack by rule 1780 (line 7393):
-> $$ = nterm lifetime_opt ()
Stack now 0 123
Entering state 420
Next token is token SymbolIdentifier ()
Shifting token SymbolIdentifier ()
Entering state 158
Reducing stack by rule 3 (line 747):
    $1 = token SymbolIdentifier ()
-> $$ = nterm GenericIdentifier ()
```

This trace represents the reduction stack, the sequence of parser states. Each block shows a rule being applied, where:  $$$$  produces  $\$1, \$2, \dots, \$n$

Figure 3.5: Snippet of the trace output from the design in figure 3.4.

To make use of this data, we segment the trace into *reduction blocks*, exemplified in figure 3.6. Each block compresses the essential information needed to build the PCFG. Because a LALR(1) parser processes input from terminals up to the start symbol of the grammar, reconstructing the context of each production requires inverting the sequence of reduction blocks – starting from the last and working backward.

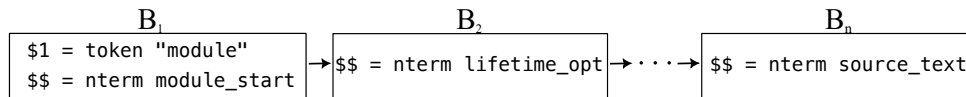


Figure 3.6: List of reduction blocks.

We feed the list of *reduction blocks* to the script depicted in figure 3.7. The entry point is `processTrace` (lines 1–8), which takes as input the list of reductions, the size of the context  $n$ , and a map to accumulate the conditional frequencies of the rules. For each reduction sequence in the list, it invokes `processReduction` (lines 11–35).

The `processReduction` function is responsible for updating the probabilistic context-free grammar. It receives the current reduction sequence, the stack of active contexts, the frequency map, and the context size `n`. First, it pops the most recent context from the stack (line 12). To maintain a fixed-size context window, we check whether the context has reached its maximum size `n` (line 14); if so, the oldest element is removed to make space for a new rule. The new rule to append is extracted from the last element in the reduction sequence (line 16), which corresponds to the current rule being reduced.

We then iterate over the remainder of the reduction sequence (excluding the last rule), treating each rule and checking whether it is a terminal or non-terminal (lines 18–24). Each rule is appended to the right-hand side (RHS) of the current production. For non-terminals, we also save a copy of the current context and push it to the stack, since additional expansions might occur along those paths.

After collecting the RHS (trimming any trailing whitespace and replacing an empty string with `%empty` if needed), we construct the left-hand side (LHS) context string by joining the rules in the current context with a tilde (`~`) delimiter (lines 26–29). Finally, we update the reduction map by incrementing the frequency count for the observed LHS  $\rightarrow$  RHS pair (lines 31–35).

This approach reconstructs the probabilistic structure of the parse by tracing how Bison reduces symbols during parsing, while maintaining an arbitrary maximum-context size. For this work, we evaluated context-size from 1 to 6 productions.

Each production rule is followed according to its probabilities. The starting symbol of the grammar has a probability of 1.0; hence, syntax generation always starts with a non-null design. If a rule  $A ::= BC$  is activated, then it creates two new nonterminals ( $B$  and  $C$ ), which will, in turn, also be activated. Each of these nonterminals might be the left-hand side of multiple productions. The choice of which production is activated depends on the probabilities associated with them. This process terminates, as terminals or the empty string are eventually produced.

The syntax tree generation process is illustrated in Figure 3.8. It begins by creating a node corresponding to the `source_text` production (line 3), which serves as the entry point of the grammar. This node is constructed using the function `createNode`, which takes the name of a production and returns an AST node of the appropriate type. Internally, this function uses a `switch` statement covering all productions in Verible’s grammar. This design allows us to apply the *visitor* pattern – a pattern that enables operations to be performed on nodes without modifying their internal structure. This pattern is central to later analysis phases, such as renaming and scope analysis (Subsection 3.2.2), type inference (Subsection 3.2.3), and code injection (Subsection 3.2.4).

The algorithm constructs the tree in a top-down, iterative manner. A stack is initialized with the root node (lines 6–7), and the tree is built by repeatedly expanding nodes popped from the stack (line 10). For each node, a grammar context is computed (line

```

1 function processTrace(reductions, n):
2     contextStack = [emptyDeque()]
3     reductionMap = map()
4     while reductions is not empty:
5         current = pop(reductions)
6         processReduction(current, contextStack, reductionMap, n)
7
8     return reductionMap
9
10
11 function processReduction(reductionSeq, contextStack, reductionMap, n):
12     context = pop(contextStack)
13
14     if length(context) == n:
15         removeFront(context)
16         append(context, treatRule(last(reductionSeq)).rule)
17
18     rhs = ""
19
20     For each r in {1...n-1}:
21         rule, isTerminal = treatRule(reductionSeq[r])
22         rhs = rhs + rule + " "
23         if not isTerminal:
24             push(contextStack, copy(context))
25
26     if trim(rhs) is empty:
27         rhs = "%empty"
28
29     lhs = join(context, "~")
30
31     if lhs not in reductionMap:
32         reductionMap[lhs] = {}
33     if rhs not in reductionMap[lhs]:
34         reductionMap[lhs][rhs] = 0
35     reductionMap[lhs][rhs] += 1

```

Figure 3.7: Script to count the frequency of productions.

13), which may depend on the node’s type and the current recursion depth. Then, a list of productions is sampled based on the grammar and context (line 16), guiding how the node will be expanded.

Each production in the sampled sequence is processed in turn (starting at line 25). If the production is a terminal symbol – enclosed in quotes –, the quotation marks are removed, and the content is added as a terminal node (lines 26–28). When the production is the special symbol `%empty`, a terminal node with an empty string is created (lines 30–32). Otherwise, the production is normalized to lowercase for consistency. The algorithm checks whether the production is valid within the current context by consulting the grammar map (line 38). If no match is found, the production is treated as a terminal (line 40), in this case it has a valid value instead of an empty string; if it can be found on the map, it is considered an expandable non-terminal, and its corresponding node is pushed onto the stack for further expansion (lines 43 and 44). The `createNodeWithFallback`

```

1 function buildSyntaxTree(grammarMap, maxDepth, generator):
2     # It starts from the 'source_text' root and expands nodes by sampling productions
      from a given grammar distribution.
3     root = createNode("source_text")
4     setParent(root, null)
5
6     stack = emptyStack()
7     push(stack, root)
8
9     while stack is not empty:
10         current = pop(stack)
11
12         # Compute the grammar context for this node (may depend on depth)
13         context = getNodeContext(current, maxDepth)
14
15         # Choose a sequence of productions randomly based on the grammar and context
16         productions = chooseProductions(grammarMap, context, generator)
17
18         children = emptyVector()
19
20         # Recompute context for the children
21         childContext = getNodeContext(current, maxDepth - 1)
22         if childContext is not empty:
23             appendSeparator(childContext)
24
25         for each production in productions:
26             if production is a terminal (enclosed in quotes):
27                 clean production (remove quotes)
28                 child = createTerminalNode(" " + production + " ")
29
30             else if production is "%empty":
31                 # Represents an empty production
32                 child = createTerminalNode("")
33
34             else:
35                 # We need to always assume lowercase strings later on the code
36                 normalized = lowercase(production)
37
38                 if grammarMap does not contain (childContext + production):
39                     # We reached a terminal
40                     child = createNodeWithFallback(normalized, " " + production + " ")
41                 else:
42                     # Expandable nonterminal, push it to stack
43                     child = createNodeWithFallback(normalized, production)
44                     push(stack, child)
45                     setParent(child, current)
46                     append(children, child)
47                 setChildren(current, children)
48     return root
49 }

```

Figure 3.8: Function to build the syntax tree from a grammar distribution.

function is a way to enable dealing with productions that we do not support; in this case, the function returns a generic *Node* object. Each created node is assigned its parent (line 45), collected as a child, and finally attached to the current node (line 47). This process continues until the stack is empty, resulting in a fully expanded abstract syntax tree rooted at *source\_text* (line 48), and at the end of syntax generation, we obtain a skeleton of a Verilog design, as Example 2 illustrates.

**Example 2** *Figure 3.9 (a) shows an example of a design produced by exercising the probabilistic grammar. Notice that this design is not valid, among other things, because every symbol is referred to as a placeholder.*

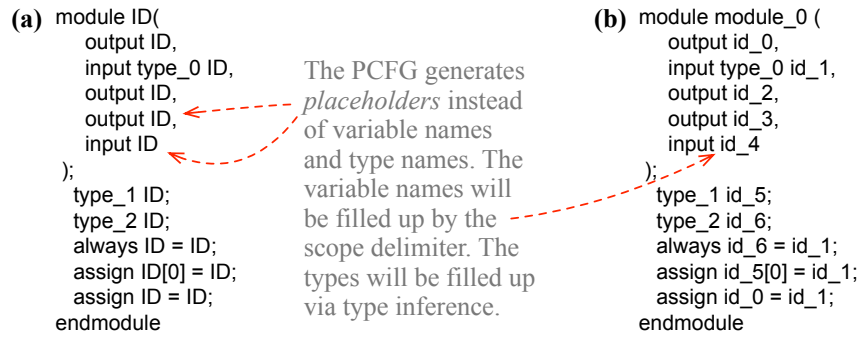


Figure 3.9: Skeleton generation and symbol renaming.

### 3.2.2 Variable Renaming and Scope Creation

The design in Figure 3.9 (a) contains only placeholders where symbol names are expected. In the next stage of the code generation, a “scope delimiter” replaces these placeholders with variable names. Renaming uses a set of “in-scope” variables and follows three rules: (i) The declaration of a placeholder is renamed with a new symbol  $s$ , and  $s$  is inserted into the set of in-scope elements associated with the current scope. (ii) Uses of a placeholder are randomly replaced with symbols, respecting only their direction. Input identifiers should never appear on the left-hand side of assignments, and the contrary holds for output signals. (iii) Once the scope delimiter leaves a scope region, it removes from the set of in-scope elements the variables declared within that region. Example 3 shows how these rules are applied in practice.

**Example 3** Figure 3.9 (b) shows the effect of applying renaming on the skeleton earlier discussed in Example 2. Variable usages, initially represented by generic ID names, are replaced with any of the user-defined symbols  $ID_0$  to  $ID_6$ , which are in-scope.

#### 3.2.2.1 Renaming Variables

As previously mentioned, each production in the grammar is represented by its own dedicated class, all of which inherit from the generic `Node` class (see Figure 3.10). This base class defines the common data members and methods used in all the derived classes. Method definitions are omitted for brevity.

Each `node` instance maintains a list of its children and a pointer to its parent, enabling upward and downward traversal of the syntax tree. The `element` member typically stores the name of the production and is later used during name resolution and code generation. The `type` field, of enum type `NodeType`, identifies the specific production represented by the node. This field is especially helpful when searching for specific children in a list of generic `node` pointers, as it eliminates the need for explicit type casting.

```

1  class Node {
2      List<Node> children
3      Node *parent
4
5      String element
6
7      NodeType t = NodeType::BASE_NODE
8      ...
9  }
```

Figure 3.10: Generic node class from which all production classes inherit. It defines the common data members and methods.

The renaming process makes use of the structured nature of the *syntactic tree*. We implement this operation using the *visitor* pattern. During tree traversal, we maintain a stack of active scopes. Each scope keeps track of the identifiers that are currently visible and valid for use.

In SystemVerilog, scopes are introduced by constructs such as modules, programs, and interfaces, as well as by named blocks, functions, and classes. When visiting a node, we use the current scope to determine the available identifiers. If the node is a symbol identifier (e.g., in an assignment), we select one randomly from the list of visible identifiers.

Figure 3.11 illustrates this process. Within the *assign*  $ID = ID$  construct, only identifiers declared in the current scope or inherited from the parent scope are considered valid. From this list, we randomly select `ID_0` and `ID_1`. Since the assignment does not involve a declaration (i.e., it does not introduce a new identifier), renaming is performed by directly modifying the `element` member of the corresponding `node`, as shown in Figure 3.10. Introducing a new identifier would not impose additional restrictions. The only required step would be to instantiate and register it in the list of available identifiers for the current scope. Once renaming is complete for a given scope, we discard any identifiers declared within it and remove the scope from the stack, restoring the previous context.

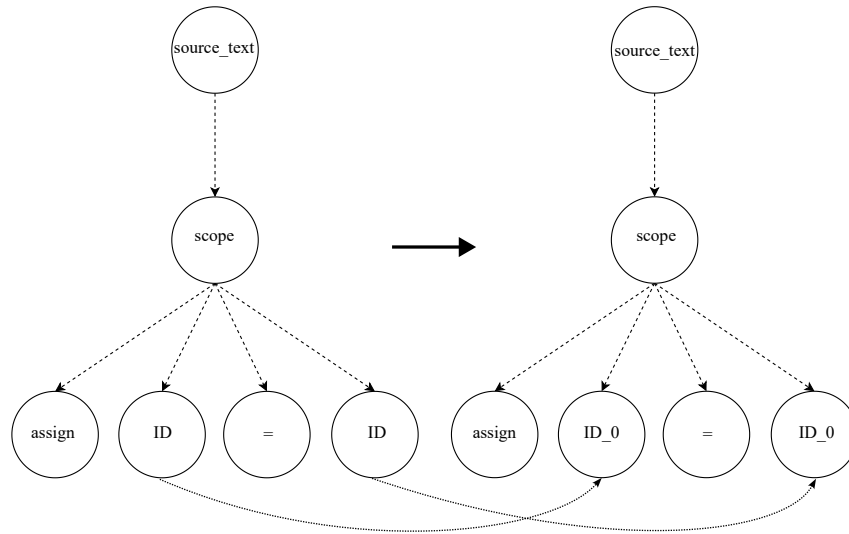


Figure 3.11: Identifier renaming through syntax tree traversal.

### 3.2.2.2 Dealing with Instantiable Namespaces

An *instantiable namespace* is a programming construct that defines a scope containing variables, functions, or other elements, where multiple independent instances of that scope can be created. Unlike non-instantiable namespaces – which provide a global or hierarchical organization of names (such as `namespace` in C++ or `package` in Java) – instantiable namespaces support multiple copies, each with its own state.

Examples of instantiable namespaces include `struct` and `union` in C, and `class` in Python. In Verilog, the `module` construct serves as an instantiable namespace. SystemVerilog extends this capability with additional constructs such as `interface`, `struct`, `union`, and `class`. Currently, ChiGen supports generating modules, unions, and both packed and unpacked structs.

Instantiable namespaces integrate naturally with the renaming process. Scope information can be inferred during downward traversal by keeping a stack of scopes. For instance, if an identifier is declared inside a struct, as shown in Figure 3.12, we can associate the struct name with the identifier during declaration. Later, when the identifier is used on the left-hand or right-hand side of an expression, we resolve the full scope path recursively, ensuring that nested scopes are handled correctly. Example 4 demonstrates this behavior.

**Example 4** Figure 3.12 (a) shows a synthetic design with references to names defined within a module. Figure 3.12 (b) shows a design with names defined within a SystemVerilog `struct`. Both constructions are supported by ChiGen.

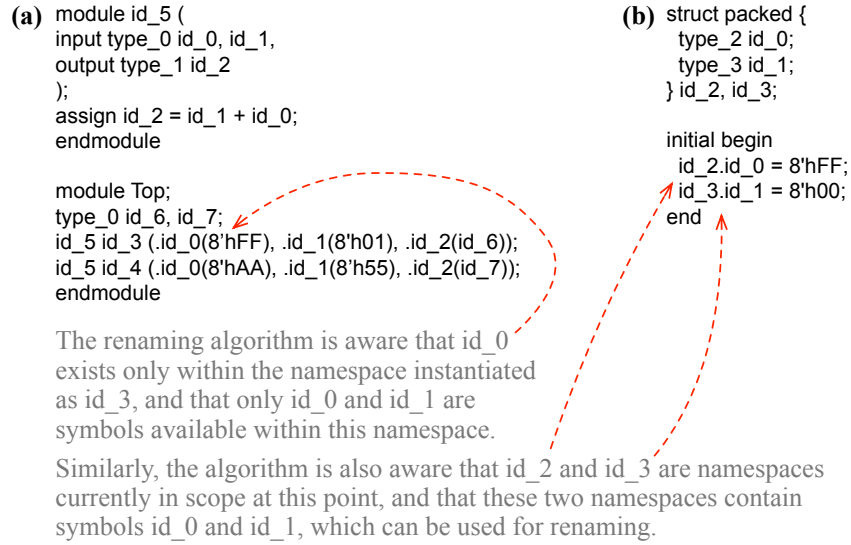


Figure 3.12: Example of synthetic designs with instantiable namespaces.

### 3.2.3 Type Inference via Unification

The scope delimiter in Subsection 3.2.2 assigns names to variables, but their types remain undefined. In the subsequent phase of code generation, a “type inference engine” deduces these types. This type inference process follows the well-known Hindley-Milner algorithm, which is widely used in languages such as SML/NJ, Haskell, and Rust. However, we adopt the two-stage formulation proposed by Sulzmann [2000]: first, we generate constraints; then, we solve these constraints through unification. Each constraint consists of a pair  $(t_0, t_1)$ , indicating that the terms  $t_0$  and  $t_1$  must share the same type. These terms may represent primitive types or open type variables (such as `type_1` in Figure 3.9). Example 5 summarizes this process, while the rest of this section details each of its two phases.

**Example 5** Figure 3.13 (a) shows the seven pairs of constraints generated for the design in Figure 3.9 (b). These pairs are produced by visiting the abstract syntax tree that describes the skeleton code. For instance, the pair  $(id\_1, id\_6)$  is produced because of the assignment *always*  $id\_6 = id\_1$  present in the skeleton. This pair indicates that the type of these two identifiers must be compatible. The result of unifying all the pairs appears in Figure 3.13 (b), where the type placeholders have been replaced with actual type names in this updated version of our running example.

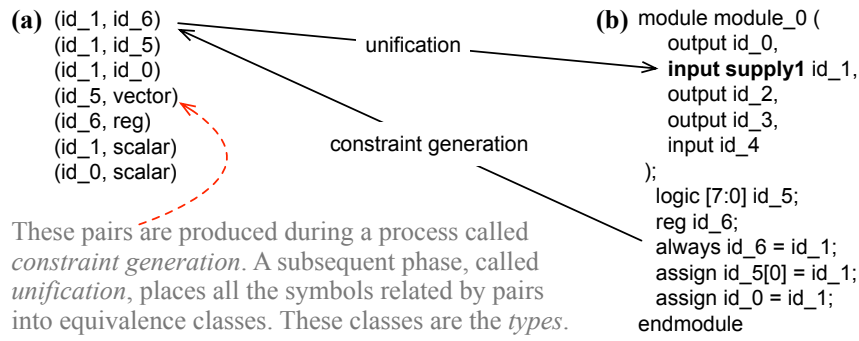


Figure 3.13: Hindley-Milner Type Inference.

### 3.2.3.1 Constraint Generation

The constraint generation process in Hindley-Milner type inference is implemented through a visitor-like traversal of the AST. The default visitor structure is shown in Figure 3.14 and represents the standard strategy used for inferring types in our system.

Type inference begins by passing a `typeId t` to each node in the tree. This identifier compresses the type information propagated from the parent node. As the visitor traverses downward, the type is forwarded to child nodes, each of which returns a set of constraints – denoted by `d1` – that must eventually be unified into a consistent set.

At every node, the constraints returned by the children are collected and propagated back to the parent. This recursive accumulation continues until the entire tree has been visited, yielding a global set of constraints that are subsequently unified during the type-solving phase.

```

1 function defaultTypeInferenceVisitor(Node node, typeId t):
2   # set of constraints generated
3   constraintSet d
4
5   for c in node.children:
6     constraintSet d1 = applyVisit(c, t)
7     d = union(d, d1)
8
9   return d

```

Figure 3.14: Default type inference visitor.

As illustrated in Figure 3.13, a type in this process falls into one of two categories:

- **Abstract type:** A symbolic placeholder used to link two or more identifiers, enforcing that they must ultimately resolve to the same type.
- **Concrete type:** A fully defined SystemVerilog type, such as `reg`, `logic`, or `wire`.

Abstract types allow the system to form equivalence classes among identifiers whose exact types are not known yet, delaying resolution until enough constraints have been collected. Concrete types, on the other hand, impose specific type restrictions on identifiers based on context.

Our inference mechanism relies on the structure of the context-free grammar. Specific grammar rules impose constraints that either bind identifiers to one another or restrict them to certain concrete types. Because these relationships are language-specific, we manually analyzed the SystemVerilog standard and encoded the constraints into our visitor logic.

Example 6 demonstrates how grammatical context informs type inference.

**Example 6** *SystemVerilog includes constructs that either bind identifiers together under a shared abstract type or constrain them to concrete types.*

*Suppose two undeclared identifiers, `id_0` and `id_1`, appear in the following contexts:*

- *The statement `assign id_0 = id_1;` establishes an abstract type relationship between `id_0` and `id_1`, indicating they must share the same type. This relationship is tracked as a constraint and resolved afterward (see Example 7).*
- *In an `always` block containing `id_0 = id_1;`, the assignment imposes a concrete constraint: `id_0` must be a `reg`, since variables assigned within an `always` block must be of a variable type. Additionally, if the expression involves a bitwise operation such as `~id_1`, the operand `id_1` must be a type that supports such operations, like `logic`, `reg`, or `wire`.*

As the visitor processes each node, it introduces type variables representing the types of identifiers and expressions at that point. It also generates a set of constraints, each expressing an equivalence or compatibility condition between two symbols—either type variables or user-defined identifiers. These constraints guarantee that operations receive operands of appropriate types and that type consistency is preserved throughout the AST. Example 7 details how these constraints are constructed.

**Example 7** *When visiting an AST node, such as `assign id_6 = id_1 + id_2;`, the constraint generation engine proceeds as follows:*

1. *Create fresh type variables for each identifier in the assignment:  $t_{id6}$  for `id_6`,  $t_{id1}$  for `id_1`, and  $t_{id2}$  for `id_2`*
2. *Create three constraints to associate each identifier with its type variable:*  
 $(id_6, t_{id6}), (id_1, t_{id1}), (id_2, t_{id2})$
3. *Create a constraint to enforce operand type compatibility:  $(t_{id1}, t_{id2})$*

4. Define two conditional constraints for result type inference:  $(t_{id1}, \text{bitvector}(N)) \Rightarrow (t_{id6}, \text{bitvector}(N + 1)), (t_{id2}, \text{bitvector}(N)) \Rightarrow (t_{id6}, \text{bitvector}(N + 1))$

These ensure that:

- The operands have the same type.
- The result follows Verilog’s bit-width extension rules.

### 3.2.3.2 Unification

Constraints are solved via a process called “*Unification*”. Unification finds a substitution of type variables by actual types that makes all the equalities hold. The final product of unification is a table – also known as an “*environment*” – that associates type variables with actual types. To build this table, the unification engine iterates through the constraints, applying known type assignments and propagating these assignments throughout the system, as Example 8 shows. If a conflict arises – such as trying to unify `bitvector(8)` with `bitvector(16)` – then the unification fails, indicating a type error. In this event, ChiGen discards the current Verilog skeleton.

**Example 8** *Continuing with Example 7, the type inference engine must solve the constraints that were produced for `assign id_6 = id_1 + id_2;`. Assume that throughout the type resolution process, the unification engine already has in the unification table the assumption that  $t_{id1} = \text{bitvector}(8)$ . This assumption, plus the constraint  $(t_{id1}, t_{id2})$  gives us that  $t_{id2} = \text{bitvector}(8)$ . Thus, substituting into the first constraint  $(id_2, t_{id2})$ , we conclude that  $t_{id2} = \text{bitvector}(8)$ . Then, applying the second constraint  $(t_{id2}, \text{bitvector}(N)) \Rightarrow (t_{id6}, \text{bitvector}(N + 1))$ , we conclude that  $t_{id6} = \text{bitvector}(9)$ .*

If insufficient constraints are available to determine the type of an identifier, then we use `wire` as the default type according to the IEEE 1800-2017 standard rule for nets declared without an explicit type. Nevertheless, even with such an expedient, some Verilog skeletons cannot undergo type inference successfully. Type inference may fail if constraints require the unification of two incompatible primitive types. When type inference fails, the skeleton is discarded, and the random seed that produced it is used as input to generate a new seed. The failure rate is influenced by the probabilistic grammar used. In the experiments described in Chapter 5, the chosen grammar yields a success rate of 50% and 75% with probabilistic contexts of length one and three, respectively.

```

1 function unify(constraints, equivalence_map):
2     for type0, type1 in constraints:
3         if type0 == type1:
4             continue
5
6         group0 = equivalence_map.get(type0, set())
7         group1 = equivalence_map.get(type1, set())
8
9         merged_group = group0.union(group1, {type0, type1})
10
11        for t in merged_group:
12            if is_fresh_type(t):
13                equivalence_map[t] = merged_group

```

Figure 3.15: Unification algorithm.

Implementing the process of *Unification* is relatively straightforward. The pseudocode shown in Figure 3.15 outlines the core logic. The algorithm operates on two main inputs:

`constraints`, which is a list of pairs (`type_0`, `type_1`), where each pair denotes a requirement that the two types must be treated as equivalent.

`equivalence_map`, a mapping from each type to a set of other types it is known to be equivalent to. This map grows throughout the unification process and defines the final type equivalence classes used during type resolution.

The algorithm works by iterating over each pair of types in the `constraints` list. For every pair (`type_0`, `type_1`), it first checks whether the two types are already the same. If they are, no further action is required for that pair. Otherwise, the sets of types currently considered equivalent to `type_0` and `type_1` are fetched from the `equivalence_map`.

These sets, along with the types `type_0` and `type_1` themselves, are merged into a new set – effectively declaring that all of them are equivalent. The algorithm then updates the equivalence mapping for each type in this merged set, but only if the type is fresh. Fresh types are abstract type variables that were generated dynamically during the traversal of the syntax tree. They serve as placeholders and must eventually be converted to concrete types or merged with other fresh types.

Canonical types – concrete types such as `reg`, `logic`, and `wire` – are not modified during unification. This ensures that fixed language-level semantics remain preserved.

At the end of this process, all identifiers and expressions that were restricted to be the same type will belong to the same equivalence class in the `equivalence_map`. This equivalence information is then used in the final resolution phase to assign concrete types to all abstract identifiers in the syntax tree, called “`canonicalize`”. It works as simply as removing all abstract types from the `equivalence_map` entries, keeping only the concrete types. If there is none, we fallback to the standard type (`wire`).

### 3.2.4 Code Expansion via Code Injection

To control the size of Verilog designs generated by ChiGen, we use a technique called *code injection*, following the approach introduced by Li et al. [2024]. Code injection involves combining multiple designs to create a new, syntactically and semantically valid design. In this case, a *caller* block invokes a *callee* unit using some syntax available in the target programming language. Currently, ChiGen supports the following kinds of code injection:

- **Module instantiation:** a caller module  $M_{caller}$  invokes a callee module  $M_{callee}$ .
- **Function invocation:** a caller module  $M_{caller}$  invokes a function declared inside of it.
- **Hierarchical references:** a caller module  $M_{caller}$  refers to a symbol  $R$  declared within a callee module  $M_{callee}$  or vice versa.

Example 9 shows the last two forms of injections. Example 11, at the end of this subsection, illustrates the first.

**Example 9** Figure 3.16 illustrates two forms of code injection that can be present in designs produced by ChiGen. In part (a), a function `id_9` is called within an initial block, demonstrating procedural interaction where the function’s logic depends on input `id_1`. In part (b), module `module_1` instantiates `module_0` and directly manipulates its internal wire `id_3` via hierarchical assignment (assign `id_7.id_3 = 1`), showcasing structural interaction between modules.

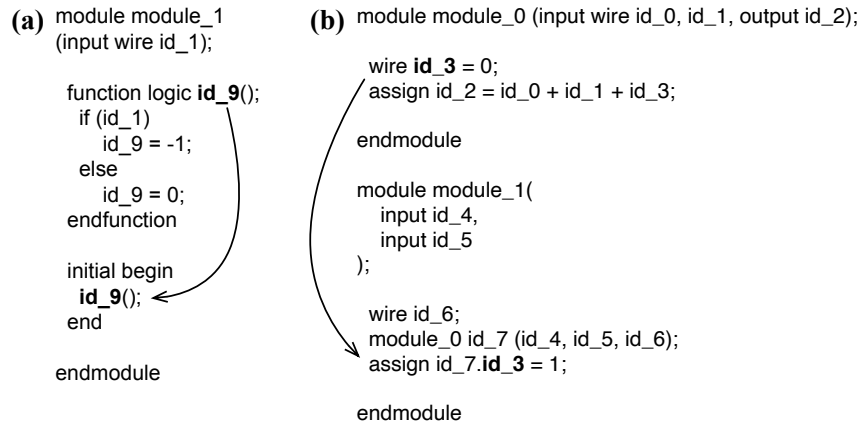


Figure 3.16: (a) Injection of function call. (b) Injection of hierarchical reference.

Figure 3.17 describes our algorithm that implements module injection (injection of function calls and hierarchical references follows a similar approach). In Li *et al.*’s

method, a new program  $P$  is built by combining two existing programs,  $P_0$  and  $P_1$ , from a real-world project. In contrast, we perform module injection interactively: as shown in Figure 3.17, we start with an empty design  $P$  and continue adding new modules to it, via the `chiGen_generate` routine, until the design reaches a preset token count,  $T$ . Notice that the `inject_module` function inserts new syntax into an existing module: this new syntax implements the instantiation rule that connects two program units, like modules, functions, or hierarchical references.

```

01 def generate_program(T):
02     P = emptyProg()
03     mod_set = chiGen_generate()
04     while P.token_count() < T:
05         C = any_module(mod_set)
06         mod_set.remove(C)
07         M, p = pick_compatible_module_primitive_gate(mod_set, C)
08         NewC = inject_module(C, M, p)
09         P.add(NewC)
10         new_set = chiGen_generate()
11         mod_set.append(new_set)
12     return P

```

We shall be accumulating the program in P.

The Verilog program that ChiGen generates contains at least one module, but usually many more; however, they do not call each other.

We inject the module instantiation for module M in the program point p that is deemed *compatible* by a reaching-definition data-flow analysis.

Figure 3.17: The algorithm that implements module injection: it injects an instantiation of M in the program point p of C.

Functions in SystemVerilog can be declared in various contexts; however, ChiGen currently supports only functions defined within modules. This restriction arises from the way the AST is constructed during the generation process – we limit generation to intra-module constructs, and as a result, function support is likewise confined to module scope.

To inject function calls into the generated code, we only insert calls into modules that define at least one function. The arguments to the function must be drawn from the set of identifiers that are visible and available at the randomly selected program point. If the module does not have a sufficient number of compatible identifiers to construct a valid function call, the injection is aborted. In this case, the module will retain a function definition but no corresponding invocation.

Automatically injecting function calls via the grammar alone is not feasible, as doing so would require context-awareness – specifically, knowledge of both the number and types of arguments expected by the function. Therefore, this process must be handled explicitly after AST generation rather than relying solely on grammar-driven expansion.

Hierarchical references introduce additional constraints during code generation. To support them, we first need to understand the module instantiation hierarchy – effectively, the call tree of the design. This is achieved by constructing a directed graph alongside

the execution of the `generate_program` algorithm (illustrated in Figure 3.17). The graph captures inter-module instantiations, including the names under which each module is instantiated.

To generate a valid hierarchical reference, we randomly select two modules – `m1` and `m2` – and attempt to find a path between them in the module instantiation graph, from `m1` to `m2` downward or upward, it is also possible to make hierarchical references by referring all instantiations starting at the top module, the top module is the root of the tree. This path represents a sequence of module instantiations that allows one module to reference identifiers declared in the other.

Once such a path is found, it is translated into a list of instance names that together form the hierarchical reference. This enables `m1` to access variables defined within `m2`, or vice versa, by following the proper instantiation chain. Example 10 demonstrates it in practice.

**Example 10** *Suppose we have a call graph as depicted in Figure 3.18. We randomly select two modules: `A` and `D`. There exists a path from `A` to `D` through `B`, giving us the call chain  $A \rightarrow B \rightarrow D$ .*

*Assume we select an identifier named `var_1` declared inside module `D`. To create a hierarchical reference to `var_1` from within module `A`, we must traverse the instantiation path. This results in the hierarchical reference:*

`assign Inst_B.Inst_D.var_1 = value;`

*Here, `Inst_B` and `Inst_D` are the instance names of modules `B` and `D`, respectively. The assigned `value` is determined based on the inferred type of `var_1`.*

*When there is no path between the two modules, for example `C` and `D`, we can use the path starting from the top module (root node):*

`assign A.Inst_B.Inst_D.var_1 = value;`

#### 3.2.4.1 Reaching Definition Analysis

Following Li *et al.*'s approach, we use the reaching definition data-flow analysis to determine where and how to inject modules into the accumulated program  $P$ . Reaching definition associates each program point  $p \in P$  with the set of variables that reach  $p$ . A variable  $v$  reaches a program point  $p$  if the program  $P$  contains a path from the definition of  $v$  until the point  $p$ , and  $v$  is not redefined along this path. We can inject a module  $M$

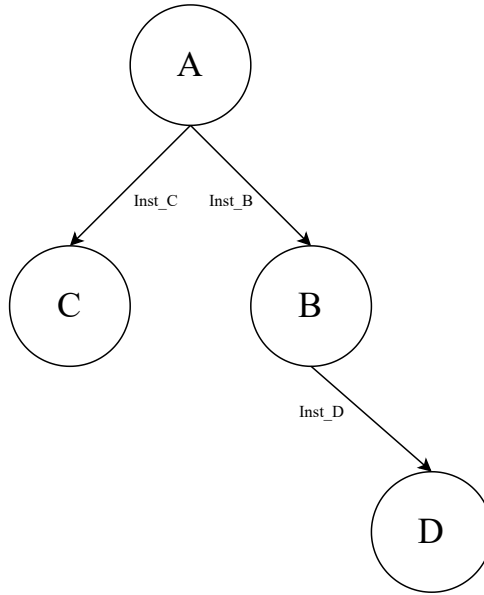


Figure 3.18: Example of a call graph for module instantiation and hierarchical referencing. Nodes represent module names, while edge labels indicate instance names.

at a program point  $p \in P$  if, and only if, for each input (respectively, output) parameter  $a$  of  $M$ , there is an input (respectively, output) variable  $v$  of equivalent type reaching  $p$ . When there are multiple such variables, we pick any of them randomly. Our module injection procedure prevents cycles in the final call graph by removing a module from the list of available modules once it is injected, as shown on Line 06 of Figure 3.17. One last observation about module injection refers to the fact that we can, at any given time, pick a Verilog primitive gate (`or`, `and`, `xor`, etc) instead of a module in `mod_set` to inject. The function `pick_compatible_module_primitive_gate` in Line 07 chooses a primitive gate or a module based on the probabilities found in the training set used in Subsection 3.2.1. Notice that primitive gates are not part of `mod_set`, since they are defined in the Verilog language; hence, they are never removed from the pool of modules available for injection.

**Example 11** Figure 3.19 shows how the reaching-definition analysis enables code injection. Variables `id_3` and `id_4` reach Line 11 in Figure 3.19 (a). These variables are compatible with the signature of `module_1`, which is part of the pool of modules available for injection. Hence, an instantiation of `module_1` is inserted in Line 11 of `module_0`.

Implementing reaching definitions follows a similar strategy to the renaming process described in Subsection 3.2.2. We traverse the AST using a visitor-like pattern while keeping a notion of lexical scopes and a list of currently available identifiers. When a scope ends, any identifiers declared within it are removed from the list.

At each point in the design, we register a set of identifiers that are considered alive—i.e., identifiers whose definitions may reach that point. The notion of program points arises naturally from the structure of the grammar. Figure 3.20 illustrates this

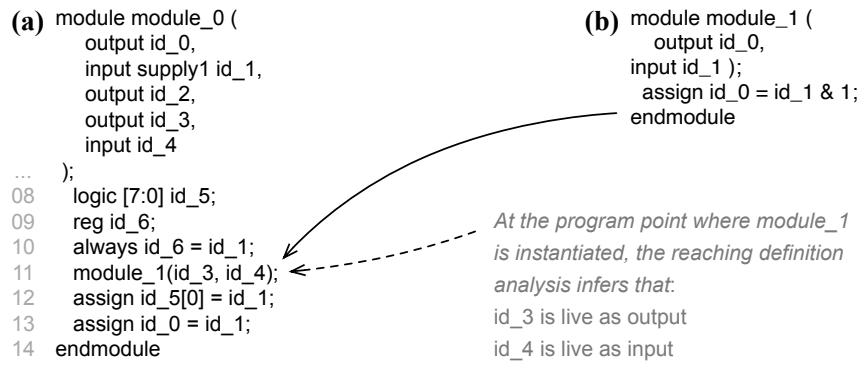


Figure 3.19: Example of a design after module injection.

concept: a program point corresponds to the closest ancestor node that represents a syntactic construct.

In the figure, the root node `source_text` eventually generates a `module_item_list`, which is a recursive production that produces zero or more `module_item` nodes. This process continues until an empty production – denoted by `%empty` in our grammar – is reached.

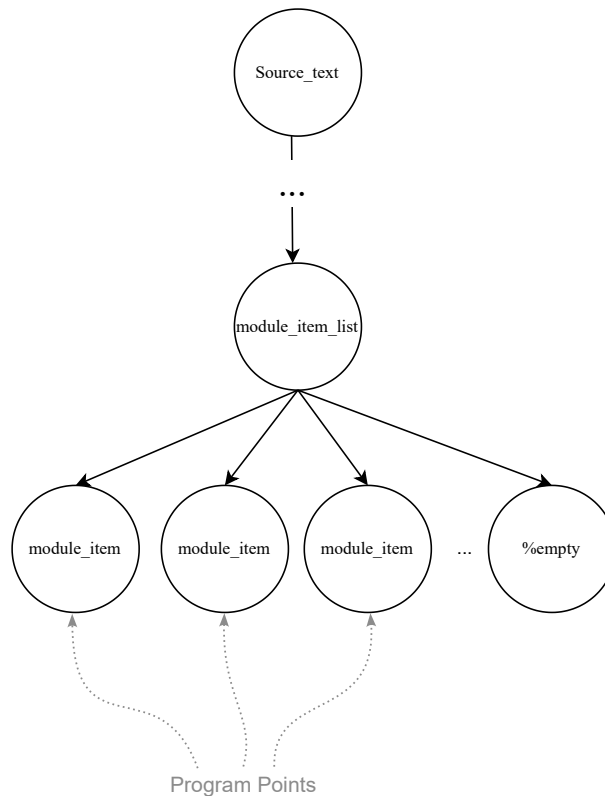


Figure 3.20: Example of an AST with the program points generated.

## 3.3 Using ChiGen

This section explains how to use ChiGen to generate SystemVerilog programs. We begin by training a probabilistic grammar in Subsection 3.3.1, followed by generating artificial hardware designs in Subsection 3.3.2. All scripts and tools discussed in this section are available in our public repository<sup>1</sup>.

### 3.3.1 Training a Grammar

Training a grammar consists of analyzing a corpus of SystemVerilog files to count the frequency of each production rule, organizing them according to the context in which they appear. This data is then used to build a probabilistic context-free grammar to guide the generation process.

ChiGen relies on Verible’s Verilog parser for analyzing source files. It is important to use our modified version of Verible<sup>2</sup> rather than the mainstream version, as our branch includes changes that expose production rule names compatible with ChiGen.

We provide a helper script, `run_parser_count Productions.sh`, which automates the parsing and grammar extraction:

```
./scripts/run_parser_count Productions.sh ./database ./verible-verilog-syntax
grammar.json context-size
```

The arguments are:

- `./database`: Path to the directory containing the SystemVerilog training files.
- `./verible-verilog-syntax`: Path to the Verible parser binary.
- `grammar.json`: Output file for the generated grammar.
- `context-size`: Positive integer that defines how much contextual information to use (e.g., 1, 2, or 3).

Training the grammar from scratch can take hours, depending on the size of the input corpus. As an alternative, pre-trained grammar files are available in the `json` folder of our repository.

---

<sup>1</sup><https://github.com/lac-dcc/chimera>

<sup>2</sup><https://github.com/joaovam/verible>

### 3.3.2 Generating Designs

Once the grammar has been trained (or a pre-trained one has been selected), ChiGen can be used to generate SystemVerilog designs. The syntax for running ChiGen is:

```
./build/Chimera ./grammar.json 1
```

In this example:

- `grammar.json` is the path to the file containing the probabilistic grammar.
- `1` indicates the number of programs to generate.

The generated design will be written to `stdout`. ChiGen also supports many flags that can be used to dump debug information and a verbose mode; this output is written to `stderr`.

If a pre-trained grammar is being used, simply replace `grammar.json` with the desired file. For example:

```
./build/Chimera ./json/1gram.json 1
```

This command will generate one SystemVerilog program using the pre-trained grammar in `./json/1gram.json`.

## 3.4 Design Examples

In this section, we present three examples of SystemVerilog modules generated by ChiGen. These examples display the variety of constructs that ChiGen can produce – ranging from simple arithmetic expressions to complex hierarchical references and scoping rules. It is important to highlight that a design generated by ChiGen is considered valid when it passes both the `analyze` and `elaborate` phases in Cadence Jasper. However, due to differences in language coverage and implementation, not all EDA tools may accept every valid construct.

**Example 1: Basic Signal Assignments and Parameter Use** The first example, shown in Figure 3.21, illustrates a straightforward module that combines wire declarations, logic vectors, and parameter-driven assignments. This design exercises fundamental aspects of the language, such as constant expression evaluation, indexed signal assignments, and type promotion rules in SystemVerilog.

```

1 module module_2;
2
3     wire id_1 = 0;
4
5     wire id_2 = id_1;
6
7     logic [7:0] id_3, id_4;
8
9     parameter id_5 = 1;
10
11     assign id_3[0] = id_2 - 1;
12
13 endmodule

```

Figure 3.21: SystemVerilog design with arithmetic expressions and vector assignments.

Although this example appears simple, it can still expose weaknesses in toolchains. In particular, it forces the analysis of type-correct arithmetic, such as the assignment `id_2 - 1` to a specific bit within a logic vector – a scenario that has been shown to cause type width inference issues in certain tools.

```

1 module module_1 (input id_0);
2     wire id_1, id_2, id_3;
3
4     reg id_4;
5
6     assign id_1 = ~id_0;
7
8     module_0 modCall_1 (
9         id_1,
10        id_2,
11        id_3
12    );
13 endmodule
14
15 module module_0 (
16     input tri1 id_0,
17     output wor id_1,
18     output tri0 id_2,
19 );
20     wire id_3;
21
22     localparam id_4 = 1;
23
24     assign id_3 = -1;
25
26     assign module_1.id_4 = 0;
27 endmodule

```

Figure 3.22: SystemVerilog design with hierarchical reference and cross-module assignment.

**Example 2: Inter-Module Connections and Hierarchical Assignments** The second example, depicted in Figure 3.22, demonstrates a more complex scenario involving two interconnected modules. The design uses a range of signal types, including `tri1`, `wor`, and `tri0`. Beyond traditional module instantiation, this example features a hierarchical assignment: the module `module_0` assigns a value to a variable declared in its parent module, `module_1` (`module_1.id_4`). Since `module_1` instantiates `module_0` (lines 9–13), the assignment at line 27 is syntactically valid, yet unconventional. This example is mainly effective for assessing whether tools correctly implement hierarchical name resolution and scoping semantics.

```

1  module module_1 (
2      id_2,
3      id_3,
4  );
5      inout uwire id_3;
6      input wire id_2;
7      bit id_8;
8
9      assign id_3 = 1;
10
11     function void id_9;
12         logic id_10;
13         begin : LABEL_0
14             id_8 = id_2;
15         end
16     endfunction
17
18     always @* id_9();
19 endmodule

```

Figure 3.23: SystemVerilog design featuring non-ANSI port declarations and function scoping.

**Example 3: Functions, Scoping, and Non-ANSI Declarations** The third example, illustrated in Figure 3.23, combines non-ANSI port declarations, function definitions, and explicit scoping via named blocks. The module features an `inout` port, a function with an internal labeled block (`LABEL_0`), and the assignment of signals both inside and outside procedural blocks. The call to the function `id_9` (line 18) is injected into the code after generating the skeleton of the design, as demonstrated in Subsection 3.2.4.

This type of construct is useful for testing a tool’s ability to handle mixed port declaration styles, scoping within function bodies, and the structural correctness of procedural logic. Such combinations, while syntactically valid, are often under-tested in industrial flows and can reveal both parsing and synthesis inconsistencies.

This chapter presented an overview of ChiGen’s current design generation process. It began with each phase involved in the generation process, followed by a brief guide on

---

how to use ChiGen to generate designs, and concluded with real examples produced by the tool. The workflow starts with the expansion of grammar rules from the probabilistic context-free grammar, followed by a renaming phase based on the identifiers in scope. Type inference is then applied to filter out designs that are likely to be semantically invalid. Finally, a code injection phase introduces additional constructs to enhance diversity in the generated output.

## Chapter 4

# ChiBench: The Training Set

The probabilistic grammar used in the open-source distribution of ChiGen (see Subsection 3.2.1) was trained on a dataset of 10,000 designs. These designs were selected from ChiBench, a larger collection of 50,000 designs curated specifically for training ChiGen. To extract this subset, we sorted the ChiBench designs by size, measured in terms of the number of tokens. From this ordered sequence, we selected the 5,000 designs immediately below the median and the 5,000 designs immediately above it. This approach helps avoid outliers – designs that are either too small or too large – while keeping training times manageable. This chapter details the methodology used to construct the ChiBench collection.

In order to build ChiBench, we have mined designs from open-source GitHub repositories, using their REST API. We build a list of candidate Verilog repositories from it. The 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. Assigns a unique name to each `.v` file, based on its repository and its local path;
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 a predefined number of files is reached.

## 4.1 Curating the Data

After we have copied the necessary number of Verilog files from GitHub, we proceed to select valid designs. 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 syntactic analyzer. At this stage, if Verible’s parser cannot build an abstract syntax tree for a file, we discard it. Example 12 illustrates one such situation.

**Example 12** *The design in Figure 4.1, which specifies an 8-bit counter, will be filtered out by the syntactic filter. It contains a missing semicolon in line 7. Such syntactically invalid files might occur in the mining process, 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 syntactically invalid because it misses a semicolon at the end of line 7

Figure 4.1: Specification filtered out by syntactic verification.

Once we remove any syntactically invalid designs, we use Jasper’s HDL semantic analyzer to filter out any semantically invalid designs. Notice that Jasper’s HDL analyzer also rejects invalid syntax. However, Jasper’s HDL analyzer is more computationally expensive than Verible’s because it also considers semantic analysis, being more adherent to the Verilog language standard. Consequently, to reduce the number of designs sent for semantic analysis, we chose to filter out syntactically invalid designs before using Jasper. Example 13 better explains the semantic analysis’s role.

```

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**)

Figure 4.2: Verilog specification that fails the semantic test.

**Example 13** *Figure 4.2 shows an example of a design 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. Thus, this design is invalid because it is trying to assign a value to **data** inside an **always** block, but **data** is declared as an output wire. In Verilog, nets cannot be assigned inside an **always** block; only **reg** or **logic** variables can be written values in procedural contexts.*

---

This chapter introduced ChiBench, the dataset developed specifically to support the training of the open-source version of ChiGen. ChiBench is composed of a curated collection of Verilog designs from open-source repositories on GitHub and was used to perform all the experiments presented in Chapter 5.

# Chapter 5

## Evaluation

This chapter evaluates the following research questions:

**RQ1:** How diverse are the designs that ChiGen generates?

**RQ2:** How do ChiGen’s designs compare with real-world codes in terms of the coverage that they enable in typical EDA tools?

**RQ3:** Which kinds of bugs can be uncovered via ChiGen-enabled fuzzing?

**RQ4:** What is ChiGen’s latency, measured in terms of Verilog designs produced per second?

**RQ5:** What is the effectiveness of the different techniques listed in Section 3.2 to increase the percentage and size of valid Verilog designs?

**Baselines** We have used 10,000 ChiBench designs (Chapter 4) to train ChiGen’s probabilistic grammar. We compare ChiGen with Verismith [Herklotz and Wickerson, 2020], TransFuzz [Solt and Razavi, 2025] and VlogHammer [Wolf, 2021], which are other fuzzer collections. VlogHammer always generates the same 3,000 designs.

### 5.1 RQ1 – Diversity

This section evaluates the *syntactical diversity of features* of ChiGen-generated designs. In Section 5.2, we will examine—indirectly—their semantic diversity by assessing the coverage they enable when used as input files for EDA tools. We measure syntactical diversity as the number of unique production rules in the Verible grammar required to parse these files.

**Discussion** Figure 5.1 shows the syntactical diversity across populations of various sizes of ChiGen designs. The figure counts unique production rules, meaning that multiple occurrences of the same rule (e.g., `Decl ::= “wire” Id “;”`) within a population are counted only once. We observe that as the number of generated designs grows, the number of unique production rules used also increases, approaching the number found in ChiBench, our ground truth. In contrast, VlogHammer, Verismith, and TransFuzz exercise significantly fewer production rules.

In populations of 10,000 designs, ChiBench exercises 406 unique production rules, increasing to 456 in the complete dataset. Among the fuzzers, Verismith exercises 179 and TransFuzz, 151. VlogHammer – which is limited to 3,000 designs – uses 137 unique productions. ChiGen’s performance varies slightly with the size of the probabilistic context  $K$ : for  $K = 1, 2, 3, 4, 5, 6$ , we observe 377, 362, 362, 357, 360, and 350 unique productions exercised, respectively.

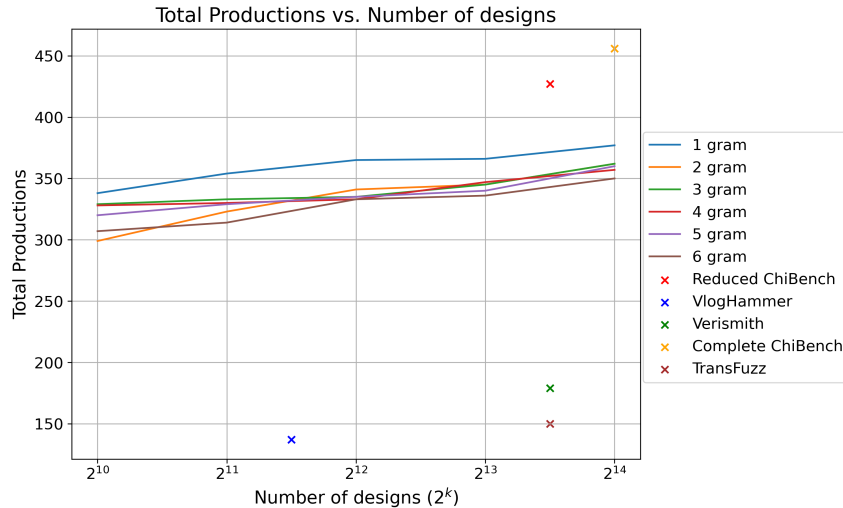


Figure 5.1: Syntactical diversity of ChiGen designs, measured as the number of unique production rules in the Verilog grammar exercised when parsing a population of generated files.

Because ChiGen exercises a broader range of production rules than other fuzzers, the programs it generates exhibit greater diversity in both the number and types of tokens used. To illustrate this, Figure 5.2 reports the number of unique 4-grams observed in a population of 10,000 designs. A 4-gram here refers to a sequence of four token types, such as `ID → less_than → int → semi_colon`. The six populations generated by ChiGen, each using a different probabilistic context, closely match the diversity seen in ChiBench—a benchmark composed of real-world Verilog designs. In contrast, the outputs from other fuzzers show a substantially smaller number of distinct 4-grams. This analysis highlights ChiGen’s ability to produce highly diverse code, approaching the variability found in real Verilog programs and significantly surpassing that of existing fuzzers.

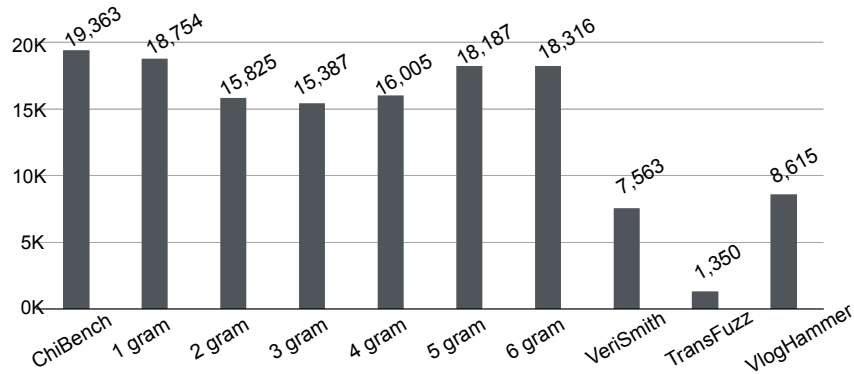


Figure 5.2: Diversity of ChiGen designs, measured as the number of unique 4-grams: the number of different sequences formed by four kinds of tokens.

## 5.2 RQ2 – Coverage

Code coverage in software testing measures the percentage of code executed during testing. A test set is considered better than another if it enables higher coverage. This section evaluates the coverage of ChiGen designs using two metrics: *branch coverage* and *line coverage*. Branch coverage measures how many of the possible branches or decision points in a program have been executed by a set of test cases. In the context of conditional statements like `if` or `case`, branch coverage checks whether each possible path (or “branch”), such as both the true and false paths of an `if` statement, has been tested. A coverage of 100% indicates that every instruction of the binary program was fetched at least once during the execution of the test case. Line coverage counts how many lines in the source code of the tested program were executed during the test. This metric tracks the execution of executable lines in code, pointing it focuses on lines that contain actual instructions that are executed during runtime. Comments or blank lines do not count toward coverage. Both metrics are measured via Clang’s source-based code coverage feature (available in Clang 14.0.0).

**Discussion** In each of the eight charts in Figure 5.3, the same trend is observed: ChiBench achieves the highest coverage, followed by ChiGen, VeriSmith, TransFuzz, and VlogHammer, in that order. The difference between ChiGen and the other fuzzers is noticeable; in some cases, such as Verible’s parser, it results in nearly twice the coverage.

Another noteworthy observation is the consistent and gradual improvement in coverage with ChiGen designs. Indeed, coverage has not stabilized in any of the charts, although it reaches a plateau of small gains fairly quickly in Verible’s formatter. However, this trend is not observed in the other fuzzers, which all seem to converge to a fixed percentage of code coverage after generating around 6,000 samples. This observation corroborates the findings in Section 5.1, which suggest that ChiGen designs are more

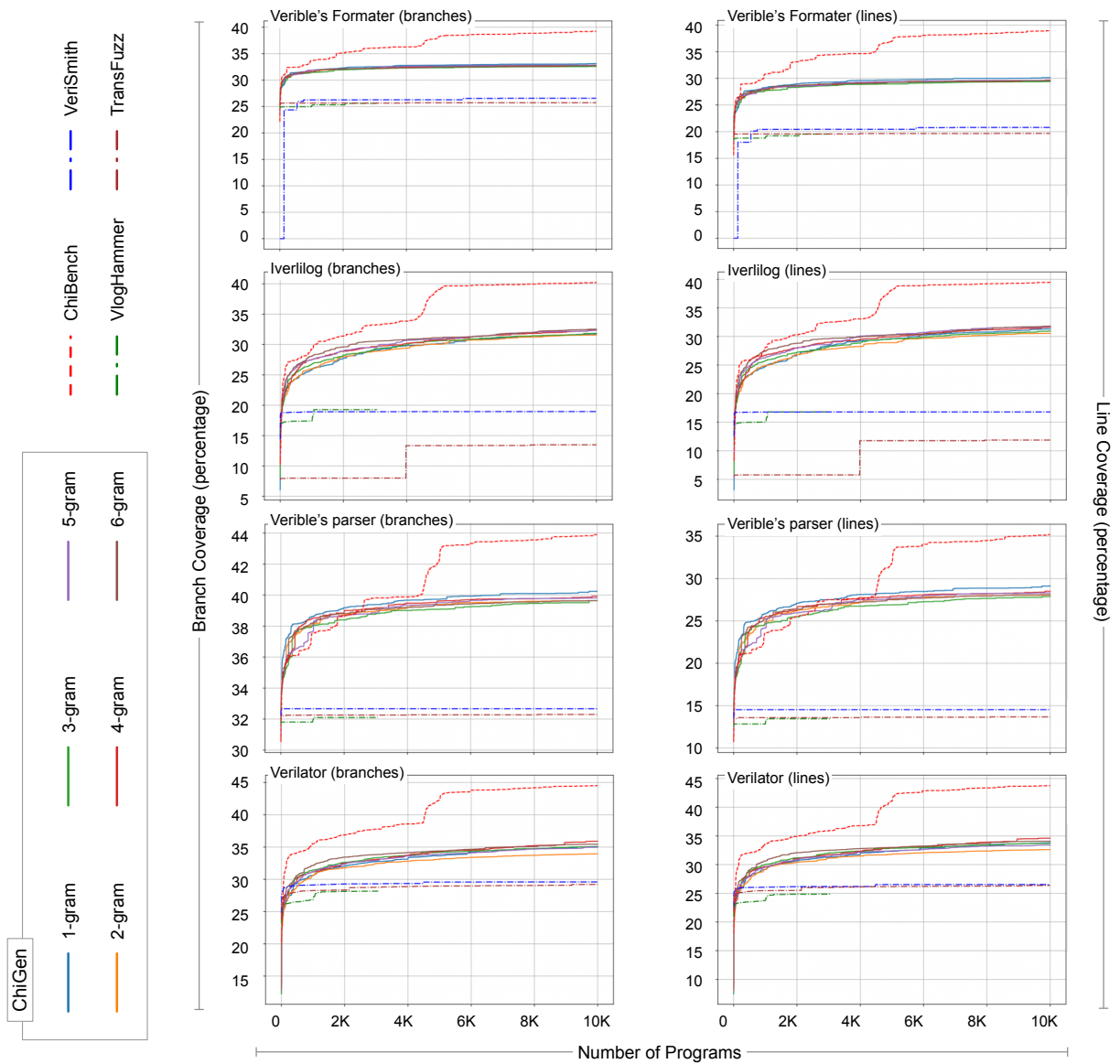


Figure 5.3: Branch coverage obtained by testing Verible’s syntactic analyzer, formatter, Iverilog, and Verilator.

diverse than those produced by the other fuzzers.

The coverage achieved through human-made ChiBench designs outperforms that of synthetic benchmarks in every experiment. One reason for this superior performance is the SystemVerilog syntax. Currently, ChiGen’s support for SystemVerilog is limited: it does not generate features such as classes and objects, fork-join, wait fork, dynamic and associative arrays, constrained random stimulus generation, clocking blocks, or interface types, for example. This limitation is not dependent on the context of probabilities used in the probabilistic grammar: these features were intentionally omitted. Incorporating them into ChiGen is more a matter of development priorities than an inherent theoretical challenge. Over time, additional SystemVerilog features will be integrated into the tool, and we hope that the open-source community will contribute to expanding ChiGen’s capabilities in this area.

## 5.3 RQ3 – Bugs

ChiGen was designed as a qualification tool for the Jasper Formal Verification Platform and has been used for this purpose within Cadence Design Systems’ development methodology. The effectiveness of ChiGen in this context is classified information. Nevertheless, ChiGen has been used to test a number of open-source EDA tools. These tests have revealed issues, many of which were reported and acknowledged in public forums. Table 5.1 lists some of these issues that our group is aware of.

Table 5.1: Issues discovered in various Verilog tools via tests using ChiGen designs.

| Issue | Tool                     | Description                                                                               |
|-------|--------------------------|-------------------------------------------------------------------------------------------|
| 2159  | Verible’s Obfuscator     | The tool crashes when reading a design that only contains the pragma directive.           |
| 2181  | Verible’s Parser         | The tool crashes instead of reporting syntax errors related to instantiation type.        |
| 2189  | Verible’s Code Formatter | The tool crashes with syntactically valid input.                                          |
| 2233  | Verible’s Parser         | The tool incorrectly accepts Verilog code with mismatched program and endmodule keywords. |
| 5276  | Verilator                | The tool crashes with signal 9 on a very large design.                                    |
| 5311  | Verilator                | The tool crashes when using time assignments.                                             |
| 5312  | Verilator                | The tool crashes when calling a function created in a “generate” block.                   |
| 5865  | Verilator                | The tool crashes when passing inout ports to primitive gates.                             |
| 1174  | Icarus Verilog           | The tool crashes when assigning to parameters in a procedural block.                      |
| 1225  | Icarus Verilog           | The tool freezes when computing invalid infinite loop.                                    |
| 4598  | Yosys                    | The tool crashes while simplifying design.                                                |
| 2359  | Verible’s Code Formatter | The tool fails to parse a dash in front of an unary operation ( <code>- -1</code> ).      |
| 2364  | Verible’s Code Formatter | Fails to parse parameter declaration without qualifier ( <code>#{id_23=1}</code> ).       |

In Chapter 3, we presented 3 examples in Figure 3.1. They were issues 2233, 2189, and 4598. To further illustrate the types of flaws that ChiGen is capable of uncovering, we present four other examples extracted from the bug list. The rest of the issues are not shown due to size restrictions.

Figure 5.4 illustrates a bug reported in the Icarus Verilog repository, where the tool enters an infinite loop while evaluating an invalid control flow structure. The loop condition, as generated, can never evaluate to true, making it impossible for the loop to end and causing the simulator to freeze.

```

1 module module_0;
2     genvar id_6;
3     for(genvar id_8 = -1; -1; id_6 = -1);
4 endmodule

```

Figure 5.4: Issue 1225 in Icarus Verilog: The tool freezes when computing an invalid infinite loop.

Figure 5.5 highlights an issue identified in Verible’s parser. In this case, the parser crashes when it encounters syntactically invalid module instantiation code rather than reporting an error. The expression `id_0var_0 (var_3)` deceives the parser into interpreting it as an anonymous instance. However, the succeeding tokens `var_1`, `var_2` conflict with this interpretation by suggesting named instance syntax, causing the parser to terminate

unexpectedly. Our research group addressed this issue by submitting a modification to Verible’s SystemVerilog grammar, which was later merged into the official project.

```

1 module m();
2     id_0var_0 (var_3), var_1, var_2;
3 endmodule

```

Figure 5.5: Issue 2181 in Verible’s parser: The tool crashes instead of reporting syntax errors related to instantiation type.

Figure 5.6 illustrates an issue encountered in Verible’s formatter, where the tool misinterprets the expression `--1`. Instead of correctly parsing it as two consecutive unary minus operators, the lexer incorrectly merges `--` into a single invalid token. This misinterpretation causes the formatter to crash during the lexing or parsing stage.

```

1 module module_0 ( ) ;
2     assign id_9 [ -1 - -1 ] = 0 ;
3 endmodule

```

Figure 5.6: Issue 2359 in Verible’s formatter: The tool fails to parse dash in front of unary operation (`--1`).

Figure 5.7 presents a design that triggers a failed assertion in Icarus Verilog. In this case, Icarus is unable to maintain correct lexical scoping during module instantiation, resulting in the runtime error: `assert: pform.cc:1478: failed assertion lexical_scope == cur_module`. The root cause of the issue lies in the incorrect use of a parameter on the left-hand side of an assignment, which constitutes undefined behavior according to the SystemVerilog standard. Despite the code being invalid, the expected behavior for a synthesis or simulation tool would be to reject the design gracefully, rather than terminate execution through an assertion failure.

```

1 module module_0 #(
2     parameter id_3 = 32'd50,
3     parameter id_9 = 32'd40
4 ) ();
5
6     case ((1))
7         1: begin
8             if (id_3) begin
9                 else begin
10                     end else begin
11                         if (1)
12                             id_3 = id_9[1];
13
14                     end
15                 end
16             end
17         endcase
18 endmodule

```

Figure 5.7: Issue 1174 in Icarus Verilog: The tool crashes when assigning to parameters in a procedural block.

To complement Table 5.1, this section demonstrates how ChiGen compares to other fuzzers in terms of its ability to reveal crashes in EDA tools. To this end, we have carried out bug-finding campaigns on different open-source tools: Verible (v0.0-3808)’s obfuscator, Yosys v0.45, and Verilator Release 159. In this process, we had to compile each tool with and without AddressSanitizer (Asan) [Serebryany et al., 2012]. Each campaign consists of producing a population of 3,000 designs – 500 for each production context using ChiGen – and submitting these designs to each EDA tool. We classify as an “issue” any situation where a random Verilog design causes either a segmentation fault or a failed assertion.

**Discussion** Table 5.2 summarizes our findings. When compiled with AddressSanitizer, ChiGen revealed 754 crashes in Yosys compared to 747 from both Verismith and VlogHammer and 742 from TransFuzz; 52 against a single crash from both tools in Verilator and no crash from TransFuzz; and 766 in Verible’s obfuscator, where the peak was 771 by Verismith. Without Asan, ChiGen uncovered 47 crashes in Verilator, while Verismith and VlogHammer each found one. Finally, in Verible’s Obfuscator, ChiGen detected 719 crashes, closely trailing Verismith’s 728 and VlogHammer’s 721. Notice that we do not distinguish different crashes caused by the same issue, because that would require manually inspecting thousands of logs produced by AddressSanitizer.

|       |           | CB  | VS  | VH  | CG  | TF  |
|-------|-----------|-----|-----|-----|-----|-----|
| Asan  | Yosys     | 712 | 747 | 747 | 754 | 742 |
|       | Verilator | 0   | 1   | 1   | 52  | 0   |
|       | Obfuscate | 734 | 771 | 728 | 766 | 761 |
| !Asan | Yosys     | 0   | 0   | 0   | 0   | 0   |
|       | Verilator | 0   | 1   | 1   | 47  | 0   |
|       | Obfuscate | 0   | 728 | 721 | 719 | 0   |

Table 5.2: Crashes observed with 3,000 designs. ‘Asan/!Asan’: tool compiled with/without AddressSanitizer; CB: ChiBench; VS: Verismith; VH: VlogHammer; CG: ChiGen; TF: TransFuzz.

## 5.4 RQ4 – Latency

Latency refers to the rate at which ChiGen produces test cases over a specific period. The higher its latency, the more designs ChiGen produces. This section reports ChiGen’s latency as the time necessary to generate 100 designs, using different contexts of probabilistic productions. We set the lower limit for the number of tokens to 150.

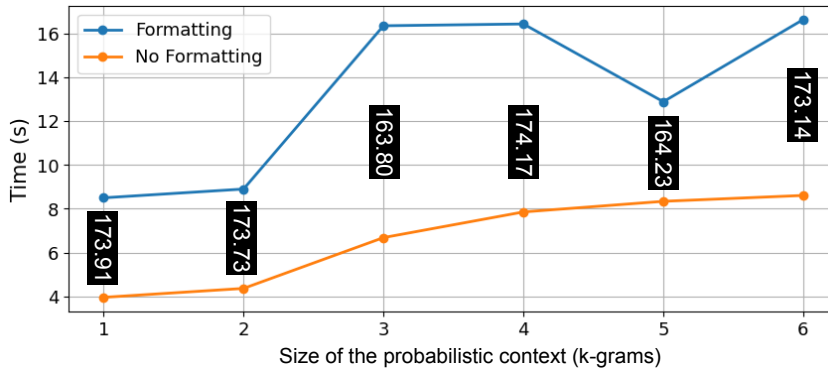


Figure 5.8: Time to generate 100 designs, setting the lower limit of tokens to 150. Numbers in boxes show the average number of tokens.

**Discussion** Figure 5.8 illustrates the latency of ChiGen, showing the time in seconds required to produce 100 designs across six sizes of probabilistic contexts. The blue line represents the time elapsed when output codes are formatted using Verible’s formatting tool. We observe a noticeable overhead introduced by formatting, which peaks at contexts of size 3, 4, and 6 grams (in the 5-gram setting, we obtained fewer tokens on average, which reduces formatting time). While formatting adds processing time, it enhances the readability of the generated code for manual analysis. In contrast, the orange line shows the time required without formatting. This line provides a more realistic picture of ChiGen’s baseline performance. Without formatting, the tool demonstrates a steady, gradual increase in processing time from 1-gram to 6-gram. Nevertheless, even in its slowest configuration – contexts of size 6 – ChiGen can still produce about 100 valid Verilog designs with at least 150 tokens each in less than nine seconds on a commodity machine.

## 5.5 RQ5 – Software Evolution

ChiGen v0.09 introduced the type inference engine described in Subsection 3.2.3, and since then, we have been tracking its effectiveness in generating valid Verilog designs. When ChiGen was announced as an open-source tool in October 2024, it was at version 0.16. Figure 5.9 illustrates the evolution of ChiGen’s capabilities over time. The figure considers a population of 1,000 designs, where valid designs are those that pass the semantic analysis performed by Jasper.

Key milestones in this evolution include: (i) Completion of the type inference engine in v0.11; (ii) First release of the module injection engine (Subsection 3.2.4) in v0.12; and (iii) Addition of an extension for varying production probabilities in v0.16. This last

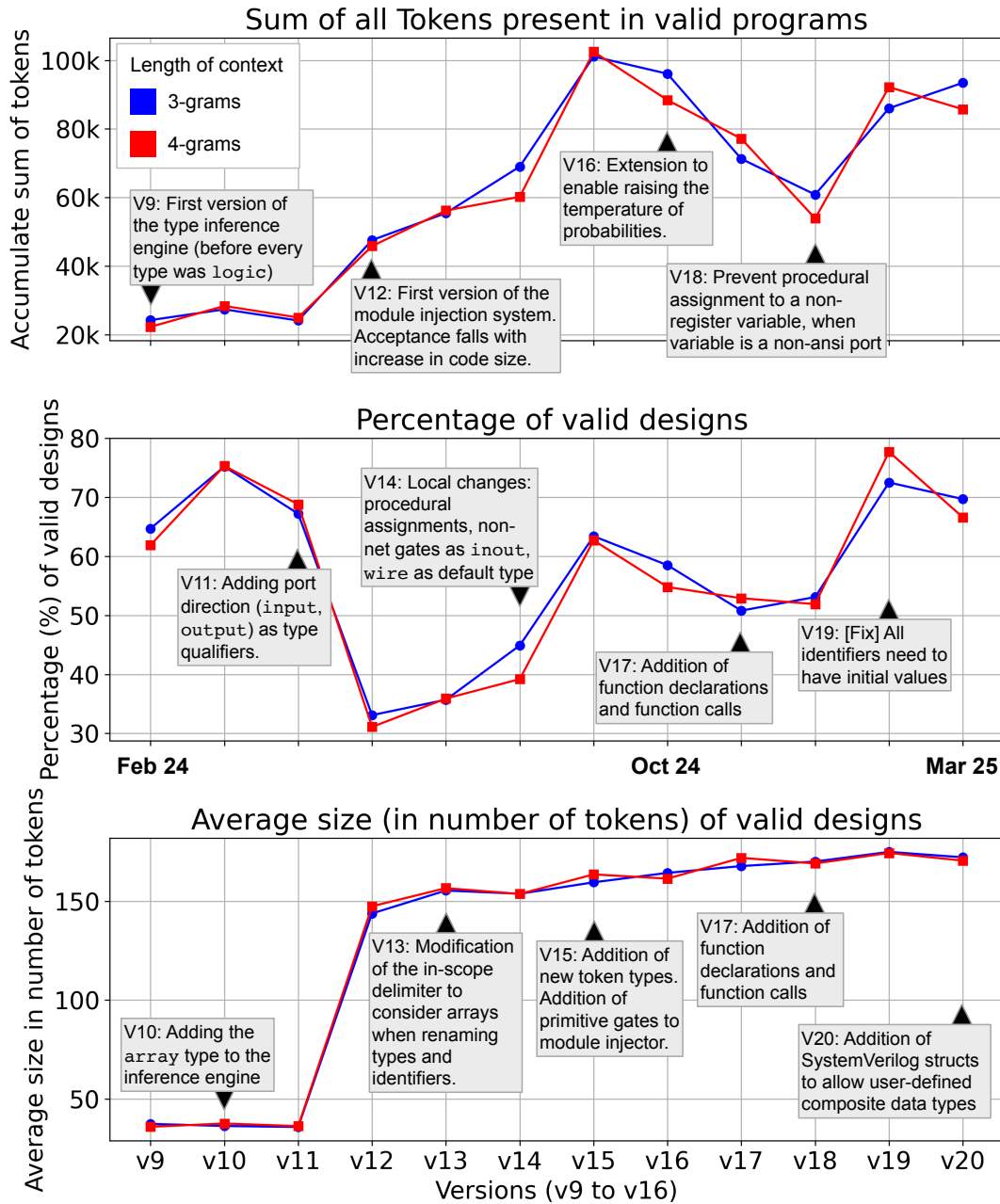


Figure 5.9: The evolution of ChiGen, until its announcement as an open-source tool in October 2024. Each dot refers to 1,000 designs. Starting in v0.12, module injection uses a lower limit of 100 tokens.

extension, not detailed in Section 3.2, enables users to “raise the temperature” of probabilities – effectively increasing the likelihood of triggering low-probability productions at the cost of slightly reducing the chances of higher-probability ones. This modification decreases the number of valid Verilog designs generated, but boosts their diversity.

Code injection (v0.12) allowed us to substantially increase the size of designs that ChiGen produces; however, it also reduced the percentage of valid codes. This reduction happened due to the probabilistic nature of the codes that ChiGen produces. Thus, the larger the text it outputs, the higher the chance that invalid specifications will emerge.

Figure 5.10 shows this trend. According to the figure, if we set the lower limit of the number of tokens to 500, then we have about 30-40% of valid designs with a probabilistic context of length two or three. The proportion is much lower with a probabilistic context of length one, as the lack of context removes much of the syntactic constraints of the Verilog grammar. Nevertheless, since v0.16, ChiGen has experienced constant evolution, and presently, the percentage of valid designs it produces is higher than pre-module injection (v0.11).

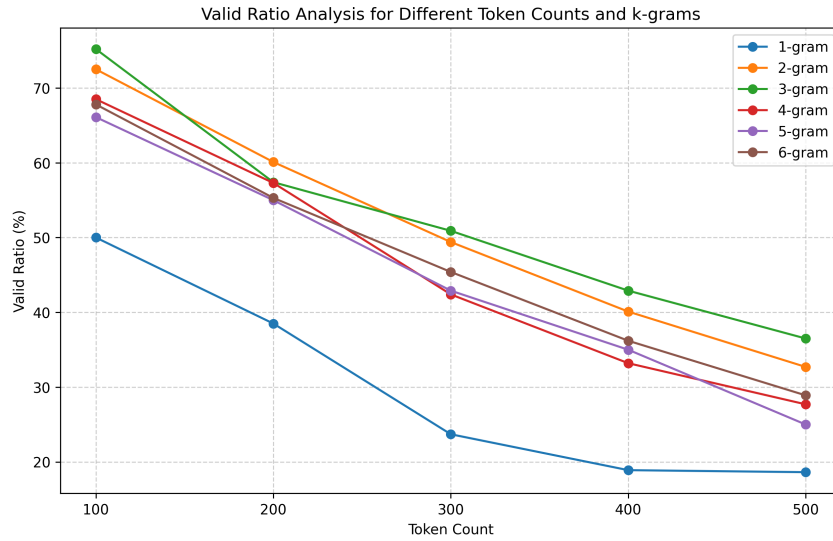


Figure 5.10: Variation of the percentage of valid designs with the lower bound on the number of tokens in the designs produced by ChiGen.

The experiments conducted in this chapter have demonstrated ChiGen’s potential as a generative tool for hardware description languages, particularly Verilog. Through a variety of test cases and usage scenarios, we evaluated ChiGen’s diversity, effectiveness compared to other tools, and performance, measured through latency, along with an overview of the versions of ChiGen.

Despite its current limitations, such as the absence of full SystemVerilog support and limited semantic control, ChiGen proved effective at generating syntactically valid and diverse Verilog designs.

Our findings suggest that grammar-based fuzzing approaches, even when limited to a specific subset of a language, can produce useful and varied inputs for testing hardware tools. Furthermore, by using a curated set of designs along with the phase of code expansion, ChiGen presents promising opportunities to enhance test case diversity while ensuring syntactic correctness.

These results provide a basis for further refinements to ChiGen’s architecture and suggest numerous potential goals for future research and development. This includes improving semantic embedding, expanding language support, and integrating learning-based mutation strategies.

# Chapter 6

## Conclusion

This work introduces the design of what we call a “bottom-up” fuzzer: a tool that generates Verilog specifications by first constructing a skeleton of Verilog syntax and then completing this skeleton by inferring names and types. While these techniques were applied in the context of Verilog, we believe that the combination of probabilistic grammars, Hindley-Milner type inference, and Li-Zhendong code injection can automate code generation for any programming language. Although none of these techniques represents a novel contribution per se, their synergistic integration as a method for implementing Verilog design fuzzers is unique.

It has always been an internal goal of this project that about 60-70% of every Verilog specification produced by ChiGen should be valid: the rest would have either syntactic or semantic faults – this decision has guided much of ChiGen’s design. As discussed in Chapter 3, the invalid outputs generated by ChiGen have uncovered zero-day bugs in several EDA tools. For instance, during a discussion on an issue ChiGen uncovered in Icarus, one of the developers commented: *“When you think about the psychology of code development, this likely makes sense. Sure, there are checks for certain invalid cases, but we can often make assumptions that our users will not get too far from valid code.”* R. [2024]”

### 6.1 Limitations

ChiGen is capable of generating SystemVerilog-like syntax; however, it does not completely support the IEEE 1800-2017 SystemVerilog standard. This limitation arises from an intended design decision by ChiGen’s developers, who chose to focus on the IEEE Standard for Verilog Hardware Description Language (Verilog-2005) Association et al. [2006]. Despite this, the ChiGen’s core is flexible. They could be extended to support other hardware description languages, including SystemVerilog, as well as general-purpose languages such as C or Java.

Additionally, ChiGen does not allow the embedding of specific semantics directly into the generated designs. To influence the characteristics of the output, users must modify the input grammar, as semantics cannot be specified.

Finally, perhaps the most fundamental restriction of ChiGen lies in the lack of direct control over the validity ratio of generated designs. Because ChiGen relies on a context-free grammar for design generation, it cannot ensure that the produced designs will be semantically valid. Securing that every possible sequence of production rules results in a valid program would require a grammar capable of expressing contextual constraints – a feature that context-free grammars lack. As a result, designing an input grammar that only produces valid designs is not feasible within ChiGen’s current architecture.

## 6.2 Future Work

As future work, multiple directions could be explored to enhance ChiGen’s flexibility and applicability. One promising development is the extension of its grammar coverage to fully support the IEEE 1800-2017 SystemVerilog standard, enabling more realistic hardware design generation for modern verification workflows. Additionally, adding semantic-aware generation mechanisms would allow users to impose functional or structural constraints on the output without manually modifying the input grammar. This would enhance ChiGen’s capabilities of generating targeted and significant test cases, especially in scenarios where specific design behaviors or corner cases must be exercised.

Another potential improvement is adapting ChiGen to support a wider variety of programming languages, including both domain-specific and general-purpose languages such as C, Java, or Rust. This could transform ChiGen from a hardware-focused generator into a versatile framework for syntactic and semantic test program generation across multiple domains.

Finally, incorporating machine learning-based techniques, such as feedback-driven grammar adaptation, could help automate and optimize the generation process, allowing ChiGen to learn from past outputs and progressively improve. For example, evolutionary or genetic search strategies could greatly improve both the efficiency and diversity of the generated designs. By generating an initial population of candidate designs and classifying them into successful (valid or interesting) and unsuccessful test cases, the valid designs could then be selected as input for the following generations. Through iterative mutation and recombination, this evolutionary process would allow ChiGen to explore a more diverse design space while refining the quality and acceptance rate of its outputs.

---

Such an adaptive feedback loop could help discover corner cases and complex patterns that are improbable of emerging from purely random or grammar-constrained generation alone, making ChiGen a more robust and intelligent test generation framework.

# Bibliography

- Ari Takanen, Jared D. Demott, and Charles Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, Inc., USA, 2nd edition, 2018. ISBN 1608078507.
- Claire Wolf. Vloghammer. <https://github.com/YosysHQ/VlogHammer>, 2021. Accessed: 2024-10-16.
- Yann Herklotz and John Wickerson. Finding and understanding bugs in fpga synthesis tools. In *FPGA*, page 277–287, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370998. doi: 10.1145/3373087.3375310. URL <https://doi.org/10.1145/3373087.3375310>.
- Flavien Solt and Kaveh Razavi. Lost in Translation: Enabling Confused Deputy Attacks on EDA Software with TransFuzz. In *USENIX Security*, August 2025. URL Paper=[https://comsec.ethz.ch/wp-content/files/mirtl\\_sec25.pdf](https://comsec.ethz.ch/wp-content/files/mirtl_sec25.pdf)URL=<https://comsec.ethz.ch/mirtl>.
- IEEE Standards Association et al. Ieee standard for verilog hardware description language (ieee 1364-2005). <http://standards.ieee.org/>, 2006.
- R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969. ISSN 00029947. URL <http://www.jstor.org/stable/1995158>.
- Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978. ISSN 0022-0000. doi: [https://doi.org/10.1016/0022-0000\(78\)90014-4](https://doi.org/10.1016/0022-0000(78)90014-4). URL <https://www.sciencedirect.com/science/article/pii/0022000078900144>.
- Martin Franz Sulzmann. *A general framework for Hindley/Milner type systems with constraints*. Yale University, New Haven, USA, 2000.
- Shaohua Li, Theodoros Theodoridis, and Zhendong Su. Boosting compiler testing by injecting real-world code. *Proc. ACM Program. Lang.*, 8(PLDI), June 2024. doi: 10.1145/3656386. URL <https://doi.org/10.1145/3656386>.
- Rafael Sumitani, João Victor Amorim, Augusto Mafra, Mirlaine Crepalde, and Fernando Magno Quintão Pereira. Chibench: a benchmark suite for testing electronic design automation tools, 2024. URL <https://arxiv.org/abs/2406.06550>.

- João Victor Amorim Vieira, Luiza de Melo Gomes, Rafael Sumitani, Raissa Maciel, Augusto Mafra, Mirlaine Crepalde, and Fernando Magno Quintão Pereira. Bottom-up generation of verilog designs for testing eda tools, 2025. URL <https://arxiv.org/abs/2504.06295>.
- Luca Amaru, Pierre-Emmanuel Gaillardon, Eleonora Testa, and Giovanni De Micheli. The epfl combinational benchmark suite, February 2019. URL <https://doi.org/10.5281/zenodo.2572934>.
- 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*, page 134, USA, 1997. IEEE Computer Society. ISBN 0818681594.
- Franco Brglez, David Bryan, and Krzysztof Kozminski. Combinational profiles of sequential benchmark circuits. In *ISCAS*, pages 1929–1934, New York, USA, 1989. IEEE.
- Krzysztof Koźmiński. Benchmarks for layout synthesis—evolution and current status. In *DAC*, page 265–270, New York, NY, USA, 1991. Association for Computing Machinery. ISBN 0897913957. doi: 10.1145/127601.127678. URL <https://doi.org/10.1145/127601.127678>.
- Kevin E. Murray, Scott Whitty, Suyu Liu, Jason Luu, and Vaughn Betz. Timing-driven titan: Enabling large benchmarks and exploring the gap between academic and commercial cad. *ACM Trans. Reconfigurable Technol. Syst.*, 8(2), mar 2015. ISSN 1936-7406. doi: 10.1145/2629579. URL <https://doi.org/10.1145/2629579>.
- Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. RTLLM: An Open-Source Benchmark for Design RTL Generation with Large Language Model, November 2023. URL <http://arxiv.org/abs/2308.05345>. arXiv:2308.05345 [cs].
- Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. VerilogEval: Evaluating large language models for verilog code generation, 2023. URL <https://arxiv.org/abs/2309.07544>.
- Ce Guo and Tong Zhao. ResBench: Benchmarking LLM-Generated FPGA Designs with Resource Awareness, March 2025. URL <http://arxiv.org/abs/2503.08823>. arXiv:2503.08823 [cs].
- Kaiyan Chang, Ying Wang, Haimeng Ren, Mengdi Wang, Shengwen Liang, Yinhe Han, Huawei Li, and Xiaowei Li. ChipGPT: How far are we from natural language hardware design, June 2023. URL <http://arxiv.org/abs/2305.14019>. arXiv:2305.14019 [cs].

- Shailja Thakur, Baleegh Ahmad, Zhenxing Fan, Hammond Pearce, Benjamin Tan, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. Benchmarking Large Language Models for Automated Verilog RTL Code Generation, December 2022. URL <http://arxiv.org/abs/2212.11140>. arXiv:2212.11140 [cs].
- Shailja Thakur, Baleegh Ahmad, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Ramesh Karri, and Siddharth Garg. Verigen: A large language model for verilog code generation. *ACM Trans. Des. Autom. Electron. Syst.*, 29(3), April 2024a. ISSN 1084-4309. doi: 10.1145/3643681. URL <https://doi.org/10.1145/3643681>.
- Mingzhe Gao, Jieru Zhao, Zhe Lin, Wenchao Ding, Xiaofeng Hou, Yu Feng, Chao Li, and Minyi Guo. AutoVCoder: A Systematic Framework for Automated Verilog Code Generation using LLMs, July 2024. URL <http://arxiv.org/abs/2407.18333>. arXiv:2407.18333.
- Ning Wang, Bingkun Yao, Jie Zhou, Xi Wang, Zhe Jiang, and Nan Guan. Large language model for verilog generation with golden code feedback. *arXiv preprint arXiv:2407.18271*, 2024.
- Yang Zhao, Di Huang, Chongxiao Li, Pengwei Jin, Ziyuan Nan, Tianyun Ma, Lei Qi, Yansong Pan, Zhenxing Zhang, Rui Zhang, Xishan Zhang, Zidong Du, Qi Guo, Xing Hu, and Yunji Chen. CodeV: Empowering LLMs for Verilog Generation through Multi-Level Summarization, July 2024. URL <http://arxiv.org/abs/2407.10424>. arXiv:2407.10424 [cs].
- Shailja Thakur, Jason Blocklove, Hammond Pearce, Benjamin Tan, Siddharth Garg, and Ramesh Karri. AutoChip: Automating HDL Generation Using LLM Feedback, June 2024b. URL <http://arxiv.org/abs/2311.04887>. arXiv:2311.04887 [cs].
- Jinwei Tang, Jiayin Qin, Kiran Thorat, Chen Zhu-Tian, Yu Cao, Yang, Zhao, and Caiwen Ding. HiVeGen – Hierarchical LLM-based Verilog Generation for Scalable Chip Design, December 2024. URL <http://arxiv.org/abs/2412.05393>. arXiv:2412.05393 [cs].
- Wenhao Sun, Bing Li, Grace Li Zhang, Xunzhao Yin, Cheng Zhuo, and Ulf Schlichtmann. Paradigm-Based Automatic HDL Code Generation Using LLMs, January 2025. URL <http://arxiv.org/abs/2501.12702>. arXiv:2501.12702 [cs].
- Yi Liu, Changran Xu, Yunhao Zhou, Zeju Li, and Qiang Xu. DeepRTL: Bridging Verilog Understanding and Generation with a Unified Representation Model, February 2025. URL <http://arxiv.org/abs/2502.15832>. arXiv:2502.15832 [cs].
- Hongliang Liang, Xiaoxiao Pei, Xiaodong Jia, Wuwei Shen, and Jian Zhang. Fuzzing: State of the Art. *IEEE Transactions on Reliability*, 67(3):1199–1218, September 2018.

- ISSN 1558-1721. doi: 10.1109/TR.2018.2834476. URL <https://ieeexplore.ieee.org/abstract/document/8371326>. Conference Name: IEEE Transactions on Reliability.
- Joobeom Yun, Fayozbek Rustamov, Juhwan Kim, and Youngjoo Shin. Fuzzing of Embedded Systems: A Survey. *ACM Comput. Surv.*, 55(7):137:1–137:33, 2022. ISSN 0360-0300. doi: 10.1145/3538644. URL <https://dl.acm.org/doi/10.1145/3538644>.
- Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Universal Fuzzing via Large Language Models. *CoRR*, January 2023. URL <https://openreview.net/forum?id=9JKEXucMSI>.
- Cristian Daniele, Seyed Behnam Andarzian, and Erik Poll. Fuzzers for stateful systems: Survey and Research Directions, January 2023. URL <http://arxiv.org/abs/2301.02490>. arXiv:2301.02490 [cs].
- Linghan Huang, Peizhou Zhao, Huaming Chen, and Lei Ma. Large Language Models Based Fuzzing Techniques: A Survey, February 2024. URL <http://arxiv.org/abs/2402.00350>. arXiv:2402.00350 [cs].
- Xiaohan Zhang, Cen Zhang, Xinghua Li, Zhengjie Du, Bing Mao, Yuekang Li, Yaowen Zheng, Yeting Li, Li Pan, Yang Liu, and Robert H. Deng. A Survey of Protocol Fuzzing. *ACM Computing Surveys*, 57(2):1–36, February 2025. ISSN 0360-0300, 1557-7341. doi: 10.1145/3696788. URL <http://arxiv.org/abs/2401.01568>. arXiv:2401.01568 [cs].
- Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. *SIGPLAN Not.*, 46(6):283–294, June 2011. ISSN 0362-1340. doi: 10.1145/1993316.1993532. URL <https://doi.org/10.1145/1993316.1993532>.
- Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Random testing for c and c++ compilers with yarpgen. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020. doi: 10.1145/3428264. URL <https://doi.org/10.1145/3428264>.
- Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. WhiteFox: White-Box Compiler Fuzzing Empowered by Large Language Models. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2): 709–735, October 2024. ISSN 2475-1421. doi: 10.1145/3689736. URL <http://arxiv.org/abs/2310.15991>. arXiv:2310.15991 [cs].
- Yuyang Rong, Zhanghan Yu, Zhenkai Weng, Stephen Neuendorffer, and Hao Chen. IR-Fuzzer: Specialized Fuzzing for LLVM Backend Code Generation, February 2024. URL <http://arxiv.org/abs/2402.05256>. arXiv:2402.05256 [cs].

- Rana Elnaggar, Brian Delgado, and Jason M. Fung. RAG-Based Fuzzing of Cross-Architecture Compilers, April 2025. URL <http://arxiv.org/abs/2504.08967>. arXiv:2504.08967 [cs].
- Chenyao Suo, Jianrong Wang, Yongjia Wang, Jiajun Jiang, QingChao Shen, and Junjie Chen. DESIL: Detecting Silent Bugs in MLIR Compiler Infrastructure, April 2025. URL <http://arxiv.org/abs/2504.01379>. arXiv:2504.01379 [cs].
- Zhihao Xu, Shikai Guo, Guilin Zhao, Peiyu Zou, Xiaochen Li, and He Jiang. A novel hdl code generator for effectively testing fpga logic synthesis compilers, 2024. URL <https://arxiv.org/abs/2407.12037>.
- Sadullah Canakci, Chathura Rajapaksha, Anoop Mysore Nataraja, Leila Delshadtehrani, Michael Taylor, Manuel Egele, and Ajay Joshi. ProcessorFuzz: Guiding Processor Fuzzing using Control and Status Registers, September 2022. URL <http://arxiv.org/abs/2209.01789>. arXiv:2209.01789 [cs].
- Deepak Narayan Gadde, Aman Kumar, Djones Lettnin, and Sebastian Simon. FuzzWiz – Fuzzing Framework for Efficient Hardware Coverage, October 2024. URL <http://arxiv.org/abs/2410.17732>. arXiv:2410.17732 [cs].
- Yike Zhou, Yanyan Jiang, and Jian Lu. Unveiling cross-checking opportunities in verilog compilers. *ACM Trans. Des. Autom. Electron. Syst.*, 30(2), February 2025. ISSN 1084-4309. doi: 10.1145/3715325. URL <https://doi.org/10.1145/3715325>.
- Yann Herklotz. Personal communication regarding the design of verismith, received on november 13th, 2024.
- Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: a fast address sanity checker. In *USENIX*, page 28, USA, 2012. USENIX Association.
- Cary R. Comment on Icarus (IVerilog) issue tracker, posted on october 11th, 2024.