Generation of various programs in Verilog/SystemVerilog for testing EDA tools

Luiza de Melo Gomes
UFMG
Belo Horizonte, Brazil
luizademelo@dcc.ufmg.br

Fernando Magno Quintão Pereira
UFMG
Belo Horizonte, Brazil
fernando@dcc.ufmg.br

***Abstract*—Testing Electronic Design Automation (EDA) tools hinges on the availability of benchmarks—programs written in Hardware Description Languages (HDLs) like Verilog, SystemVerilog, and VHDL. While benchmark collections exist, their diversity remains limited. This limitation is increasingly problematic given the growing demand for training large language models in the EDA domain. In order to address this challenge, this paper introduces enhancements in the variety of programs produced by ChiGen, a tool for synthesizing realistic Verilog designs. Originally developed to test Cadence Design Systems' Jasper™ Formal Verification Platform, ChiGen has demonstrated its capability to uncover zero-day bugs in tools such as Verible, Verilator, and Yosys. This work expands ChiGen's capabilities to include SystemVerilog constructs such as classes, interfaces, and packages, as well as formal verification primitives like assertions, sequences, and properties. These additions significantly increase both structural diversity and the semantic representativity of the generated programs.**

***Index Terms*—Verilog, Synthesis, Testing, Fuzzing.**

I. Introduction

Fuzzing is an automated testing technique that generates random, often unexpected, inputs to uncover bugs, vulnerabilities, or unintended behaviors in software. Many Electronic Design Automation (EDA) tools, including YOSYS [21], VERILATOR [14], MODELSIM™ [7], Xcelium™ [5], JASPER™[4], can benefit from fuzzers capable of automatically generating Verilog designs. Such tools enable early bug discovery, performance optimization, compliance verification, and general validation.

Existing open-source Verilog fuzzers, such as VlogHammer[20], Verismith[10], and TransFuzz[15], follow a top-down approach. They start with minimal valid Verilog syntax and expand it using various techniques, ensuring the generation of semantically valid designs. However, our experience testing the Jasper Formal Verification Platform reveals that this strict adherence to validity limits test case diversity. Interestingly, semantically invalid Verilog designs can be just as effective in identifying issues in EDA tools. Additionally, these tools often generate constructs that differ significantly from human-written Verilog, covering fewer than 40% of the production rules in the Verilog-2005 grammar (IEEE 1364-2005), as described in section V.

This paper introduces enhancements in the quantity of unique tokens and production rules to ChiGen[1], a "bottom-up" Verilog fuzzer initially developed to test Cadence Design Systems' Jasper Formal Verification Platform and released as open-source in 2024. ChiGen generates Verilog designs through a three-stage process: skeleton generation using a probabilistic grammar, mock identifier replacement with scope-compliant names and type inference using the Hindley-Milner algorithm. Our current work significantly enhances ChiGen's capabilities by expanding its token set and production rules to include modern SystemVerilog constructs like classes, interfaces, and packages, as well as formal verification primitives such as assertions.

Our experiments demonstrate that ChiGen outperforms top-down fuzzers like Verismith, VlogHammer, and TransFuzz in terms of structural diversity and bug-finding effectiveness. Since its release, ChiGen has identified at least five confirmed issues in prominent open-source EDA tools such as Yosys, Icarus Verilog [19], and Verible [6].

II. Related Work

This paper presents techniques for building Verilog fuzzers, with a specific focus on enhancing the variety of generated designs. Several other Verilog fuzzers are available as open-source tools [10], [15], [20]. Unlike our approach, these tools primarily expand a core set of Verilog syntax incrementally, ensuring each expansion results in a valid design. In contrast, ChiGen's probabilistic grammar approach enables the generation of a richer variety of tokens while maintaining syntactic correctness. During the development of ChiGen, we engaged with the authors of Verismith, gaining valuable insights into differing methodologies.

In the past two years, the emergence of large language models (LLMs) has introduced new methods for Verilog code generation [8], [12], [17], [18]. While ChiGen is not an LLM, it offers unique advantages by focusing on probabilistic grammar-based generation. Unlike LLMs, which aim to shape code toward specific semantics, ChiGen prioritizes diversity in generated tokens and production rules. Specifically, it models the probabilities of production rules as k-grams, allowing for more comprehensive exploration of Verilog's syntax space without assigning probabilities to token sequences directly.

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

ChiGen's enhanced token generation capability makes it well-suited for creating diverse Verilog benchmarks. However, it differs from existing benchmark collections [1], [2], [3], [11], [13], [16], which are static and limited to a predefined set of designs. Unlike these immutable collections, ChiGen dynamically generates new benchmarks, integrating them into its library of Verilog programs. This dynamic approach not only broadens the scope of available designs but also ensures that benchmarks can adapt to new requirements, enriching the testing landscape for hardware tools and methodologies.

III. ChiGen Overview

ChiGen is a stochastic tool for generating random Verilog programs, designed to diversify the input space for testing Electronic Design Automation (EDA) software. A high-level overview of ChiGen is shown in Figure 1. The tool takes as input the JSON grammar derived from ChiBench programs, randomly selects grammar rules based on predefined probabilities, and generates a Verilog program as output. At its core, ChiGen operates in three distinct phases, each contributing to its robust program generation process. The following sections provide a detailed explanation of these phases.

A. Syntax Generation via Probabilistic Grammars

The core of ChiGen's functionality lies in its use of n-gram language models to guide the synthesis of Verilog programs. These models require a comprehensive corpus of Verilog programs to estimate the probabilities of various language constructs, such as module declarations, always blocks, and specific operators. For this purpose, we leverage ChiBench [16], a collection of Verilog programs meticulously mined from diverse open-source repositories, serving as our primary training dataset.

To extract the production rules and their frequencies from ChiBench, we utilize Verible's parser. Verible, an open-source Verilog parsing and linting tool, provides a detailed trace of the production rules it applies during the parsing process. By analyzing these traces across the entire ChiBench corpus, we can count the occurrences of each grammar rule. Crucially, we model the probabilities of these production rules as k-grams, meaning the probability of a rule being selected depends on the context of the k preceding rules. This approach allows for a more nuanced and realistic generation of syntax, as it captures common structural patterns and dependencies found in human-written Verilog. This differs significantly from approaches that assign probabilities to token sequences directly, which can lead to less structurally coherent designs.

The collected rules and their observed frequencies, conditioned on their k-gram context, are then compiled into a JSON file. This file effectively serves as ChiGen's "probabilistic grammar." During the program generation phase, ChiGen reads this JSON file. Starting from a root production rule (e.g., "source_text"), ChiGen iteratively expands non-terminal symbols by randomly selecting a production rule from the available options, weighted by their probabilities defined in the JSON grammar and conditioned on the k preceding rules

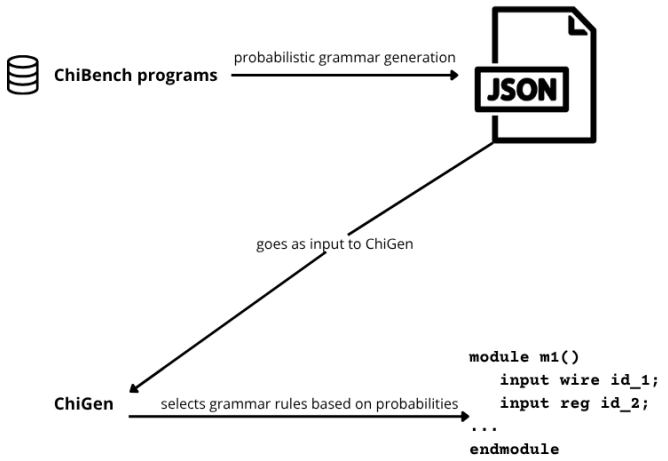

Fig. 1. ChiGen overview

in the partial syntax tree. This stochastic process ensures high variability in the generated syntax trees while maintaining a statistically representative structure based on the ChiBench corpus. The result is a syntactic skeleton of a Verilog program, ready for subsequent refinement.

In order to obtain the production rules of a program, we used Verible's parser. This is possible by using the trace of Verible's parser, which outputs production rules in the order in which they are processed. Then, we can count how many times each rule appear given the context of n preceeding rules. These rules and the frequency of each of rule is then stored in a JSON file. Once the probabilities of each construct are established in a JSON file, ChiGen synthesizes programs by assembling syntax trees guided by the language model.

B. Variable Renaming and Scope Creation

Following the initial syntax generation phase, the abstract syntax trees (ASTs) produced by ChiGen contain generic placeholders. These placeholders mark locations where specific, unique, and contextually appropriate identifiers—such as variable names, module names, port names, or instance names—are required. Proper management of these identifiers and their visibility within defined scopes is critical for generating syntactically valid and semantically meaningful Verilog and SystemVerilog programs. Incorrect naming conventions or improper scope resolution can lead to compilation errors, unintended behavioral ambiguities, or even crashes in Electronic Design Automation (EDA) tools.

In this second core phase of ChiGen's code generation, a dedicated scope management engine, which we refer to as the "scope delimiter," performs a comprehensive traversal of the generated syntactic skeleton. This traversal is typically executed in a depth-first manner, allowing the engine to meticulously identify, enter, and exit various scope regions defined by the Verilog and SystemVerilog grammar. These regions are fundamental to the language's structure and include, but are not limited to, `module` declarations, `function` and `task`

bodies, `always` blocks, `fork-join` blocks, `begin-end` blocks, `class` definitions, and `package` declarations.

The "scope delimiter" meticulously maintains a dynamic symbol table, which is conceptually structured as a stack of hash maps. Each map within this stack represents the set of `in-scope elements` (i.e., declared identifiers) for the current lexical context. As the tree traversal proceeds, the variable renaming process operates through three precise and interconnected steps:

1) **Declaration Renaming and Entry into Scope:** When the scope delimiter encounters a placeholder representing the *declaration* of a new identifier (e.g., a `wire`, `reg`, `logic`, parameter, or a new class/module name), it triggers the generation of a `new, unique symbol`. These symbols are systematically generated (e.g., `id_0`, `id_1` for variables, `module_0` for modules, `instance_0` for instances) to ensure global uniqueness within the generated program. This newly minted identifier, denoted as `s`, then replaces the generic placeholder in the Abstract Syntax Tree (AST), and `s` is immediately inserted into the symbol table that corresponds to the currently active scope. This step guarantees that every declared entity receives a distinct and resolvable name within its defined visibility.

```
// Before renaming (fragment of AST):
logic GenericIdentifier;
module GenericIdentifier (GenericIdentifier,
    GenericIdentifier);

// After declaration renaming (illustrative
    transformation):
logic id_0; // 'id_0' is a newly generated
    unique name
module module_1 (id_1, id_2); // 'module_1',
    'id_1', 'id_2' are new unique names
```

2) **Usage Resolution and Replacement:** As the traversal continues into the body of the generated code, the scope delimiter encounters placeholders that represent *uses* or references to previously declared identifiers. At each such instance, it randomly selects an already-declared symbol from the set of currently `active in-scope elements`. This selection is performed from all visible scopes, prioritizing the innermost scope to simulate standard HDL name resolution rules (e.g., local variables shadow global ones). The placeholder in the AST is then replaced with the chosen symbol. The inherent probabilistic nature of this selection introduces significant syntactic and semantic variety in how variables are referenced, which is exceptionally valuable for fuzzing. This controlled randomness can generate unusual, yet syntactically valid, connections and expressions that effectively challenge the parsing, elaboration, and semantic analysis capabilities of EDA tools.
3) **Scope Exit and Cleanup:** Upon the completion of processing a scope region (e.g., encountering an `endmodule`, `endfunction`, or `end` keyword), the scope delimiter executes a critical cleanup step. It removes from the set of in-scope elements all variables, parameters, and other identifiers that were declared specifically within that region. Conceptually, this operation corresponds to popping the current scope's symbol table off the stack. This mechanism rigorously adheres to the lexical scoping rules of Verilog and SystemVerilog, preventing erroneous references to identifiers that are no longer in scope and ensuring that names are only visible where they are legally declared. This rigorous cleanup prevents accidental cross-scope references and helps maintain the integrity of the generated design.

This variable renaming and scope creation phase is essential to ChiGen's ability to produce high-quality, diverse, and syntactically correct Verilog and SystemVerilog programs. By dynamically managing identifier uniqueness and visibility throughout the generation process, it ensures that the resulting designs are parsable and elaboratable by EDA tools. This enables effective and targeted testing of critical EDA functionalities such as symbol resolution, name binding, and hierarchical elaboration. Failures in these aspects of EDA tools, manifesting as name conflicts, unresolved references, or unexpected shadowing behavior, are common sources of bugs and can be effectively uncovered by the controlled randomness and strict adherence to scoping rules introduced during this phase.

IV. SystemVerilog and Formal Constructs

The overarching objective of this work is to significantly enhance the diversity and realism of the Verilog and SystemVerilog designs generated by ChiGen. This is achieved by strategically expanding the set of tokens and production rules that ChiGen's probabilistic grammar can utilize, thereby better reflecting the constructs found in real-world hardware designs. Accomplishing this goal involves a multifaceted approach, centered on three core tasks: systematically identifying absent or under-represented tokens and production rules within ChiGen's current generation capabilities, meticulously incorporating these missing constructions into its grammar and generation logic, and diligently correcting any erroneous programs that might arise from initial implementations of new features.

Beyond simply increasing linguistic coverage, a crucial enhancement for ChiGen is the introduction of formal verification constructs. These include SystemVerilog assert, property, and sequence blocks. For instance, ChiGen can now generate assertions such as:

```
assert property (@(*) id_1 == id_2);
```

The inclusion of these elements is vital for two primary reasons: first, they dramatically increase the language coverage, moving ChiGen beyond basic Verilog-2005 to encompass more modern and complex design patterns. Second, and perhaps more critically, they enable the generation of sophisticated test cases specifically designed to probe and challenge the formal engines of Electronic Design Automation

(EDA) tools. This allows for more targeted and effective fuzzing campaigns against formal verification platforms,

A. Addition of packages

Packages in SystemVerilog serve as containers for sharing declarations (e.g., parameters, types, functions, tasks, classes) across multiple modules or interfaces without global scope pollution. In ChiGen, modules are generated iteratively until a specified program size or complexity threshold is met. To integrate packages, the generation process was augmented: instead of exclusively generating module declarations, ChiGen now probabilistically decides whether to generate a module, package, or interface as a top-level construct.

The process for assigning unique names to generated packages mirrors that used for modules. After all primary structural units (modules, packages, interfaces) are generated, a systematic renaming pass ensures unique identifiers, typically following a pattern like module_1, module_2, package_1, interface_1, and so on. This ensures clarity and avoids name collisions within the generated design. Critically, ChiGen's scope management system was updated to understand package scopes, allowing declarations within a package to be correctly referenced (`package_name::item_name`) by other generated constructs that import or explicitly reference the package.

SystemVerilog interfaces abstract communication between design blocks, simplifying port declarations and promoting reusable verification environments. Similar to packages, interfaces are now part of ChiGen's top-level generation options. ChiGen generates interface declarations, including their modports (module ports) and interface variables. The type inference engine (Section 3.3) was extended to correctly deduce and propagate interface types when instances of these interfaces are declared within modules, ensuring that the connections adhere to SystemVerilog's strict typing rules. This enables ChiGen to produce designs that utilize more modern and complex interconnection patterns, challenging EDA tools to correctly parse and elaborate such hierarchical structures.

Packages can also be imported according to the Verilog grammar. For instance, below there is a program which produces a package and imports it:

```
package package_0;
  typedef logic id_1;
endpackage
package package_1;
  typedef logic id_2;
  import package_0::*;
endpackage
```

B. Addition of Assertions

Formal verification is an indispensable phase in modern hardware design, heavily relying on the precise specification of design behavior through constructs like assertions and properties. ChiGen's enhanced capability to generate these formal constructs significantly bolsters its utility for rigorous testing of formal verification engines within Electronic Design Automation (EDA) tools. By integrating the generation of SystemVerilog Assertions (SVAs), ChiGen can now produce highly specific and semantically rich test cases that probe the deepest layers of formal analysis.

1) Assertion Generation Strategy: ChiGen employs a sophisticated, heuristic-based approach to strategically insert assert statements into the generated Verilog/SystemVerilog code. This strategy is designed to create assertions that are contextually relevant to the surrounding design logic. A common and robust pattern in hardware description languages is the assignment of a value to a variable, which typically represents a data path, a control signal, or an intermediate computation. ChiGen intelligently identifies locations within the Abstract Syntax Tree (AST) where such assignment statements occur, for instance:

```
assign id_1 = id_2;
```

Immediately following the identification of such an assignment within the AST, ChiGen probabilistically determines whether to insert a corresponding concurrent assertion. The simplest and most fundamental form of this generated assertion checks for an equality condition:

```
assign id_1 = id_2;
assert property (@(*) id_1 == id_2);
```

This specific assertion, triggered by the @(*) event sensitivity list, validates that id_1 holds the exact value of id_2 at every point in time when either id_1 or id_2 changes. Such an assertion is invaluable for detecting unexpected value propagations, subtle functional discrepancies, or unintended temporal shifts in data. Its simplicity makes it a robust candidate for automated generation, yet its failure can indicate critical bugs in formal analysis tools or design elaboration.

The power of this dynamic assertion generation extends beyond simple variable-to-variable assignments. ChiGen's mechanism is designed to handle more complex right-hand side expressions. If the assignment involves an arithmetic, logical, or bitwise expression, the generated assertion will accurately reflect that expression, directly testing the equivalence:

```
assign id_1 = 1 + 2;
assert property (@(*) id_1 == 1 + 2);
```

This approach ensures that the generated assertions are not merely syntactically correct but also semantically tied to the surrounding design logic. This contextual relevance significantly increases their efficacy in identifying meaningful issues within formal verification tools, as they represent realistic verification challenges. This dynamic generation of assertions based on existing code patterns is a key strength, ensuring that the assertions are semantically related to the generated design logic, which consequently increases their relevance and effectiveness for formal verification. While ChiGen currently focuses on these assignment-driven assertions, future work could explore more sophisticated assertion patterns, including those involving more complex temporal operators to capture intricate design behaviors.

2) Expanded Assertion Coverage: Beyond these immediate, context-driven assertions, ChiGen also supports the generation of more elaborate assert property statements that can incorporate temporal aspects, even without explicitly defined property or sequence blocks. These assertions can directly embed temporal expressions to specify complex behavioral checks over time. This capability broadens the spectrum of formal verification scenarios that ChiGen can generate, moving beyond simple combinatorial checks to include sequential behaviors. For example, ChiGen can produce an assertion with a specific clocking event and a condition that must hold true:

```
BLOCK_0 :
assert property (id_1) @(negedge id_5);
```

C. Addition of classes

The introduction of classes in SystemVerilog significantly extended its capabilities for object-oriented programming (OOP), enabling higher-level abstraction and testbench development. ChiGen now incorporates the generation of class declarations and associated constructs, primarily focusing on static members which can be accessed without object instantiation, making them more amenable to automated generation within a fuzzer context.

1) Generation of Class Declarations: ChiGen's grammar was updated to include production rules for class declarations, including member variables (logic, int, etc.), methods (function, task), and class-specific keywords (static, local, protected). This allows ChiGen to create diverse class definitions, reflecting various class structures found in SystemVerilog.

2) Addition of calls to functions of classses: A key enhancement is ChiGen's ability to generate calls to static functions declared within a class. Static functions, unlike regular member functions, belong to the class itself rather than an instance of the class, allowing them to be called directly using the scope resolution operator (::).

ChiGen's generation process includes a phase where it traverses the partially built syntax tree. If a class definition containing a static function is found, ChiGen will probabilistically insert a call to this function in an appropriate context, such as within an always block or a task/function. The program below illustrates this concept:

```
class id_1;
    static function logic id_2(id_3);
        id_3 <=  1;
    endfunction
endclass

always @* id_1::id_2(id_3); // Call to static
    function
```

This capability challenges EDA tools' parsers and elaborators to correctly resolve static method calls and their arguments. While ChiGen currently prioritizes static function calls for their simpler integration, future developments could explore the generation of class instances, constructors, and calls to non-static methods, which would require more complex object lifecycle management within the fuzzer.

V. Evaluation

This section evaluates ChiGen's performance in addressing four key research questions:

- **RQ1:** How diverse are the designs generated by ChiGen?
- **RQ2:** What types of bugs can be uncovered using ChiGen-enabled fuzzing?
- **RQ3:** How the different techniques listed in Section III increase the diversity of Verilog designs?

a) Baselines: To train ChiGen's probabilistic grammar, we used 10,000 Verilog designs from the ChiBench collection [16]. The performance of ChiGen is then compared with other fuzzer collections, including Verismith [10], TransFuzz [15], and VlogHammer [20]. Note that VlogHammer generates a fixed set of 3,000 designs, which limits its scope of evaluation.

A. RQ1: Diversity

1) Syntactical Diversity: We evaluated the syntactical diversity of ChiGen-generated designs by analyzing the number of unique production rules in the Verible grammar required to parse these designs. The Verible grammar includes 456 distinct production rules.

a) Results: Figure 3 shows the syntactical diversity for populations of varying sizes. Unique production rules are counted only once, regardless of their frequency in the designs. As the number of generated designs increases, the diversity approaches that of ChiBench, which uses 406 unique rules for 10,000 designs. In comparison, Verismith exercises 179 rules, TransFuzz uses 151, and VlogHammer, limited to 3,000 designs, employs only 137 rules.

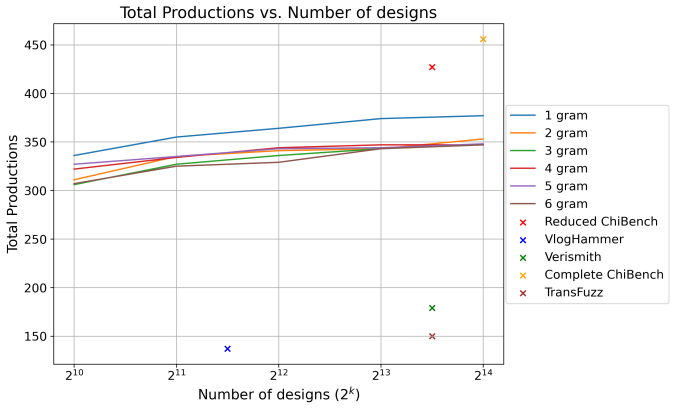

Fig. 2. Syntactical diversity of ChiGen designs, measures as the number of unique production rules in the Verilog grammar exercised when parsing a population of generated files.

ChiGen's performance varies slightly with the size of the probabilistic context K: for $K = 1, 2, 3, 4, 5, 6$, the number of unique rules exercised are 336, 355, 364, 374, and 377, respectively for 1-gram.

The results for unique token diversity parallel the positive trends observed for production rules. While a "reduced ChiBench" dataset (a specific subset of the full ChiBench corpus tailored for this particular token analysis) contains 246

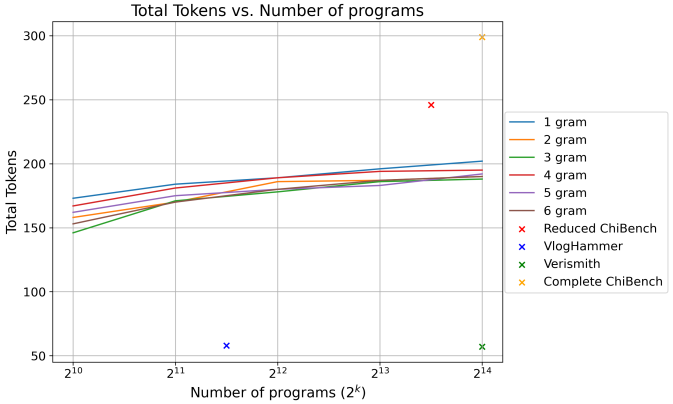

Fig. 3. Syntactical diversity of ChiGen designs, measures as the number of unique tokens in the Verilog grammar exercised when parsing a population of generated files.

unique tokens, ChiGen successfully generated designs exercising 202 unique tokens. This was observed with populations of 2^{14} (16,384) designs, specifically when generated using a 1-gram probabilistic context (i.e., $K = 1$). This demonstrates that ChiGen produces designs with a substantial variety of lexical elements, ensuring comprehensive testing of the scanner and parser components within EDA toolchains. The ability to generate such a high number of unique tokens, even with a basic 1-gram context, underscores ChiGen's foundational strength in producing rich and varied HDL code at the most granular level.

In conclusion, ChiGen's superior syntactical diversity, as evidenced by its extensive coverage of both production rules and unique tokens, is a direct consequence of its innovative bottom-up, probabilistic grammar approach. This broad and deep coverage is critical for effective fuzzing, as it enables the generation of highly varied and often unconventional test cases that are significantly more likely to uncover subtle, unforeseen bugs and vulnerabilities in EDA tools, thereby improving their overall robustness and reliability in handling the vast landscape of Verilog and SystemVerilog designs.

B. RQ2: Bug Detection

ChiGen was specifically designed to uncover bugs in the Jasper Formal Verification Platform and has been integrated into Cadence Design Systems' development methodology. Due to confidentiality agreements, the effectiveness data related to this integration cannot be disclosed.

To demonstrate ChiGen's bug-finding capabilities, we conducted extensive campaigns targeting three open-source EDA tools: Verible (v0.0-3808), Yosys (v0.45), and Verilator (Release 159). In each campaign, 3,000 designs were generated—500 for each production context—and submitted to the respective tools. The compiled designs were analyzed, and any crashes or failed assertions were flagged as issues.

a) Results: The bug-finding campaigns revealed several issues across the tested EDA tools. The following table summarizes the identified issues:

TABLE I
SUMMARY OF ISSUES IDENTIFIED IN EDA TOOLS

Issue	Tool	Description
2181	Verible	Crashes instead of reporting syntax errors related to instantiation type.
2189	Verible	Crashes with syntactically valid input.
2233	Verible	Incorrectly accepts Verilog code with mismatched `program` and `endmodule` keywords.
1174	Icarus Verilog	Crashes when assigning to parameters in a procedural block.
4598	Yosys	Crashes while simplifying program.

These findings highlight key weaknesses in the tested EDA tools. Specifically, Verible experienced several crashes, including one related to instantiation type errors, another with valid syntax, and a third with mismatched `program` and `endmodule` keywords. Additionally, Icarus Verilog and Yosys encountered crashes in scenarios involving parameter assignments and program simplifications, respectively. These results demonstrate ChiGen's effectiveness in identifying critical issues that affect the stability and reliability of EDA tools.

C. RQ3: Evolution

In this section, we examine the evolution of ChiGen by dividing it into three distinct versions, each marking a significant step in its development and capabilities.

In this section, we discuss the development trajectory of ChiGen, presenting its evolution through different versions. This chronological analysis highlights the systematic improvements implemented to broaden the diversity of generated designs and to incorporate more advanced Verilog and SystemVerilog constructs. Each version represents a significant milestone, marked by specific enhancements that collectively contribute to ChiGen's current robust state as a sophisticated fuzzer for EDA tools.

Figure 4 visually encapsulates this progression, illustrating how each iteration has incrementally contributed to the increased diversity and complexity of the generated outputs, as measured by the number of unique tokens and production rules exercised. Our analysis is based on consistent samples of 10,000 programs generated for each version, providing a reliable basis for comparison.

We delineate ChiGen's evolution through several key versions:

Initial Baseline (Pre-Version 22): This period represents ChiGen's state when this paper started, prior to the major enhancements detailed in this paper. It served as the initial point for measuring token and production rule coverage, primarily focusing on core Verilog constructs. Although capable of generating diverse designs, its coverage of modern SystemVerilog features and formal constructs was limited.

Version 22: Introduction of Packages and Minor Fixes. Version 22 marks the beginning of the significant enhancements detailed in this monograph. The primary focus of this iteration was the integration of SystemVerilog `package` constructs. As discussed in Section IV, packages provide a crucial mechanism for sharing common declarations in SystemVerilog. Their addition required modifications to ChiGen's grammar and its scope management system to correctly handle package declarations and their usage. This version also

incorporated several minor bug fixes that addressed previously identified syntactic inaccuracies in the generated programs. The impact of these changes was a noticeable expansion in the types of structural organization ChiGen could produce, laying the groundwork for more complex designs.

Version 23: Initial Integration of Classes. A major undertaking in Version 23 was the initial introduction of SystemVerilog `class` constructs. This was a significant step towards enabling the generation of more complex, object-oriented test cases. However, the initial integration of classes proved to be particularly challenging. The complexities associated with class syntax, scope rules, and type interactions led to a dramatic decrease in the percentage of syntactically valid designs. This phenomenon is a common characteristic during the integration of highly complex language features into fuzzer grammars, as subtle interactions can lead to widespread parsing errors if not meticulously handled. Although groundbreaking in its scope, this version highlighted the need for rigorous error correction and refinement in subsequent iterations.

Version 24: Major Syntax Error Resolution and Stability Improvement. Version 24 was primarily dedicated to addressing the major syntax errors caused by the initial addition of classes in Version 23. This involved a thorough review of the newly added class-related production rules, refinement of their probabilities, and significant debugging of the type inference and variable renaming phases to correctly handle class-specific contexts. The concerted effort in this version led to a substantial increase in the number of valid designs, indicating a greater maturity in ChiGen's ability to produce well-formed SystemVerilog code that incorporates classes. This improvement was crucial for ensuring that a higher proportion of generated programs could be successfully parsed and processed by EDA tools, thus increasing the efficiency of bug detection campaigns.

Version 25: Addition of Static Class Method Calls and Further Refinements (Current Version). The most recent iteration, Version 25, represents the current state of ChiGen. A key enhancement in this version was the specific integration of calls to `static` functions within classes. While this feature significantly expanded the functional coverage of SystemVerilog classes, its intricate nature, particularly concerning argument passing and return types, caused a slight, albeit manageable, decrease in the immediate percentage of perfectly valid designs. This transient reduction reflects the ongoing challenge of integrating advanced language features into a stochastic fuzzer while maintaining high levels of syntactic correctness. However, the benefit of generating more realistic and challenging test cases for EDA tools, especially those related to static class member resolution and formal verification, far outweighs this minor fluctuation. This version also includes continued minor refinements to existing grammar rules and probabilistic models based on ongoing feedback and further analysis of the ChiBench corpus.

The data presented in Figure 4 visually reinforce this narrative. The progression illustrates a consistent effort to expand the breadth of ChiGen's output, steadily increasing the total number of unique tokens and production rules generated. This measured and iterative approach to development allows ChiGen to progressively cover more of the Verilog and SystemVerilog language, ensuring that it remains a cutting-edge tool for rigorous EDA tool testing.

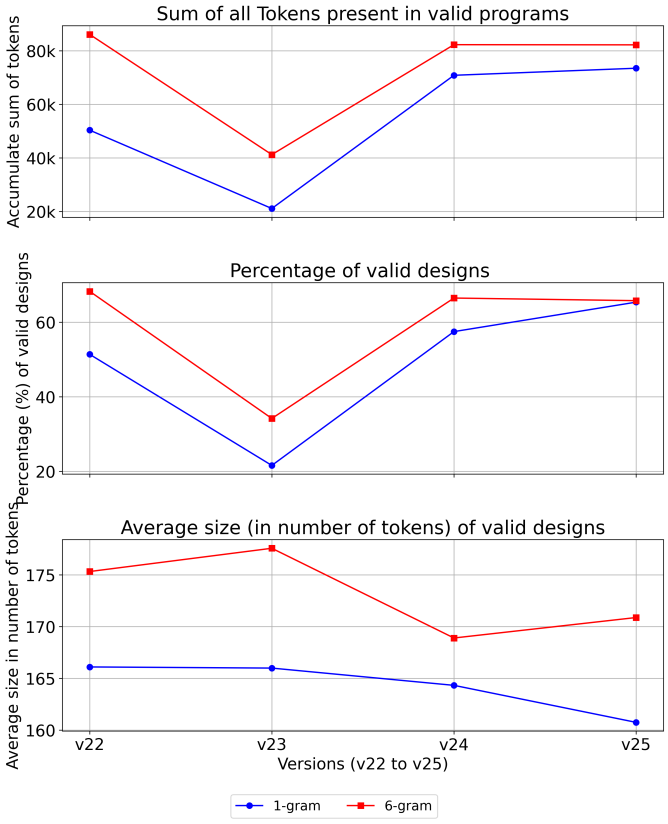

Fig. 4. Evolution of the number of tokens and productions.

VI. Future Work

Despite ChiGen's significant achievements in generating diverse and effective Verilog and SystemVerilog designs for testing Electronic Design Automation (EDA) tools, several key areas remain open for further exploration and potential enhancement. Notably, unlike some fuzzers such as Verismith [9], which prioritize the generation of 100% syntactically and semantically correct designs, ChiGen's current probabilistic, bottom-up approach means it does not consistently produce entirely correct programs. This inherent trade-off, which contributes to ChiGen's ability to expose obscure bugs, also points to specific directions where its capabilities could be expanded by subsequent research.

One primary area for future investigation involves refining ChiGen's generation mechanisms to potentially increase the rate of syntactically and semantically correct outputs without unduly sacrificing the structural diversity that is its hallmark. This might entail more sophisticated post-generation validation and correction passes, or deeper integration of semantic checks during the generation process itself, to preemptively avoid certain classes of errors. Achieving higher correctness rates

could broaden ChiGen's applicability, particularly for EDA tools that are less tolerant of malformed inputs or require consistently valid designs for advanced analysis.

Beyond merely improving correctness, a wealth of SystemVerilog features remain to be explored for integration into ChiGen's generation framework. For instance, concerning SystemVerilog classes, the current implementation primarily focuses on static members and basic class structures. Future work could implement the generation of non-static class fields, enabling object instantiation with constructors (new), and the modeling of basic inheritance. Such additions would allow ChiGen to create more complex object-oriented test environments, which are increasingly vital in modern verification methodologies.

Similarly, within the domain of formal verification, while ChiGen effectively generates assert property statements, the comprehensive generation of named property and sequence blocks, along with their varied instantiations, represents a significant unaddressed area. The framework is also capable of incorporating the generation of `covergroup` and `coverpoint` constructs for functional coverage, which would enhance its utility for testing coverage analysis tools. Furthermore, exploring the generation of designs that leverage SystemVerilog's interface and modport features for more realistic inter-module communication presents another opportunity to deepen its language coverage.

Another potential direction for future development involves investigating mechanisms for more targeted design generation. Although ChiGen's stochastic nature excels at exploring the design space broadly, subsequent research could explore ways to guide the generator to produce designs with specific characteristics, such as a minimum number of modules, specific class types, or a particular density of assertions. This added control would facilitate more focused testing campaigns for particular tool features or verification methodologies.

By addressing these challenges and expanding its linguistic breadth, ChiGen's framework offers a robust foundation for continued development, promising further insights into the stability and robustness of next-generation EDA tools.

VII. Conclusion

This paper presented significant advancements to ChiGen, a "bottom-up" Verilog fuzzer, through the expansion of its token set and improvements in program correctness. ChiGen generates Verilog designs by constructing a syntactic skeleton, inferring names and types, and injecting additional constructs using a probabilistic grammar and the Hindley-Milner type inference. Although these techniques are well-established individually, their integration within ChiGen represents a unique approach to Verilog fuzzing.

A key design philosophy of ChiGen is balancing valid and invalid program generation, with approximately 70% of its outputs being valid Verilog designs. This deliberate inclusion of semantically and syntactically invalid programs has proven effective in uncovering zero-day bugs in several EDA tools, as highlighted in Section III. The addition of new tokens to ChiGen has further enhanced its ability to generate structurally diverse and realistic Verilog programs, making it a more powerful tool for testing EDA tools.

The addition of SystemVerilog and formal verification constructs increases the relevance of ChiGen for modern hardware development workflows, positioning it as a comprehensive benchmark generator for structural and formal EDA testing.

Acknowledgments

This project is sponsored by Cadence Design Systems. Additionally, Luiza de Melo acknowledges the support of Rafael Sumitani, João Victor Amorim, Mirlaine Crepalde and Augusto Mafra.

References

[1] Luca Amaru, Pierre-Emmanuel Gaillardon, Eleonora Testa, and Giovanni De Micheli. The epfl combinational benchmark suite, February 2019.

[2] 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.

[3] Franc Brglez, David Bryan, and Krzysztof Kozminski. Combinational profiles of sequential benchmark circuits. In *ISCAS*, pages 1929–1934, New York, USA, 1989. IEEE.

[4] Cadence Design Systems, Inc. Jasper formal verification platform. https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification.html, 2025. Accessed: 2025-06-23.

[5] Cadence Design Systems, Inc. Xcelium logic simulator. https://www.cadence.com/en_US/home/tools/system-design-and-verification/simulation-and-testbench-verification/xcelium-simulator.html, 2025. Accessed: 2025-06-23.

[6] CHIPS Alliance. Verible. https://github.com/chipsalliance/verible, 2025. Accessed: 2025-06-23.

[7] Xilinx Edition II. Modelsim®. 1990.

[8] 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. arXiv:2407.18333.

[9] Yann Herklotz. Personal communication regarding the design of verismith, received on november 13th, 2024.

[10] 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.

[11] Krzysztof Koźmiński. Benchmarks for layout synthesis—evolution and current status. In *DAC*, page 265–270, New York, NY, USA, 1991. Association for Computing Machinery.

[12] Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. Verilogeval: Evaluating large language models for verilog code generation, 2023.

[13] Kevin E. Murray, Scott Whitty, Suya 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.

[14] Wilson Snyder. Verilator 4.0: open simulation goes multithreaded. In *Open Source Digital Design Conference (ORConf)*, 2018.

[15] Flavien Solt and Kaveh Razavi. Lost in Translation: Enabling Confused Deputy Attacks on EDA Software with TransFuzz. In *USENIX Security*, August 2025.

[16] 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.

[17] 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 2024.

[18] 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.

[19] Stephen Williams and Michael Baxter. Icarus verilog: open-source verilog more than a year later. *Linux Journal*, 2002(99):3, 2002.
[20] Claire Wolf. Vloghammer. https://github.com/YosysHQ/VlogHammer, 2021. Accessed: 2024-10-16.
[21] Clifford Wolf, Johann Glaser, and Johannes Kepler. Yosys-a free verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, volume 97, 2013.