

Bottom-Up Generation of Verilog Designs for Testing EDA Tools

João Victor Amorim Vieira
Cadence
Belo Horizonte, Brazil
jvamorim@cadence.com

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

Rafael Sumitani
Cadence Design Systems
Belo Horizonte, Brazil
srafael@cadence.com

Raissa Maciel
Cadence Design Systems
Belo Horizonte, Brazil
raissa@cadence.com

Augusto Mafra
Cadence Design Systems
Belo Horizonte, Brazil
augusto@cadence.com

Mirlaine Crepalde
Cadence Design Systems
Belo Horizonte, Brazil
mirlaine@cadence.com

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

Abstract—Testing Electronic Design Automation (EDA) tools rely on benchmarks: designs written in Hardware Description Languages (HDLs) such as Verilog, SystemVerilog, or VHDL. Collections of benchmarks for these languages are typically limited in size. 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, ChiGen, was originally developed to test the Jasper[®] Formal Verification Platform, a product by Cadence Design Systems. Now, 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 of it: (i) generation guided by probabilistic grammars, (ii) type inference via 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.

Index Terms—Verilog, Synthesis, Testing, Fuzzing.

I. INTRODUCTION

Fuzzing [23] 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[®], XCELIUM[™], JASPER[®], and Design Compiler[®] 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 [27], Verismith [8], and TransFuzz [20]. 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 Verification Platform suggests that the requirement to produce only valid Verilog designs limits the diversity of their test cases. As outlined in Section II,

semantically invalid Verilog specifications can be equally effective as valid ones in uncovering issues in EDA tools. Additionally, these tools often produce syntactic constructs that are very different from those found in human-written code. For instance, as detailed in Section V-A, these tools cover fewer than 40% of the production rules in a grammar of Verilog-2005 (IEEE 1364-2005) [2].

Contributions of This Work: This paper 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 III, 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 over the benchmark suite detailed in Section IV. Second, ChiGen replaces mock identifiers with names that respect scoping rules. Third, it applies the Hindley-Milner type inference algorithm [21], commonly used in functional programming, to infer the types of variables. Finally, it employs a technique recently proposed by Li *et al.* [14] to combine Verilog modules, functions, and references, achieving any predefined number of tokens.

As discussed in Section V, 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 Section II, 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 permissible licenses. This benchmark suite, ChiBench, is a contribution in itself and has been used for tasks beyond training ChiGen, as discussed in Section IV.
- **Probabilistic Grammar:** As detailed in Section III-A, ChiGen can be trained on 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 static scoping rules are obeyed (Section III-B) and that types are respected (Section III-C).
- **Code Injection:** ChiGen incrementally integrates generated components into more complex designs, linking modules via bindings, instantiations, function calls, or hierarchical references (Section III-D). As new modules, functions, and references are generated, they become available for insertion into the ongoing design.

The ChiGen fuzzer and the ChiBench suite of Verilog designs are publicly available under the GPL 3.0 license and can be retrieved at <https://github.com/lac-dcc/chimera>.

II. OVERVIEW

This section illustrates the usage of ChiGen with the three designs seen in Figure 1. These files were automatically produced by ChiGen. All these designs have uncovered an issue in some EDA tool, which was reported and acknowledged. Throughout this paper, 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 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.

```
(a) module module_0();
endprogram

(b) module module_0(id_1);
output logic
signed id_1;
endmodule

(c) localparam id_1 = id_1;
module module_0();
endmodule
```

Fig. 1. (a-b) Designs that uncovered issues in VERIBLE. (c) Design that uncovered issues in YOSYS.

Tool Crashes on Valid Syntax: The code in Figure 1 (b) is valid; however, the port `id_1` is declared as `output logic signed`. This syntax is not typical in traditional Verilog but is acceptable in SystemVerilog. Nevertheless, this specification causes a segmentation fault in Verible’s parser.

Tool Crashes on Invalid Semantics: The design in Figure 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. This module causes infinite recursion in Yosys, eventually forcing a crash, as the tool runs out of stack memory.

In addition to the three examples described in this section, 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 paper, we will only discuss issues that have been publicly reported.

III. BOTTOM-UP FUZZING

ChiGen works in four phases, which Figure 2 shows. The rest of this section describes each of these steps.

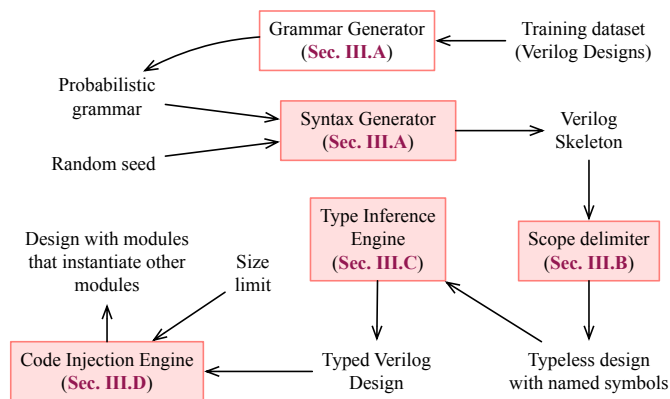


Fig. 2. Overview of ChiGen’s modus operandi.

A. 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 2, ChiGen is distributed with a grammar generator. This tool receives a training set (a collection of Verilog designs). 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 Section IV.

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 K rules (a K -gram), resulting in "sequence-aware" probabilities. We limit the probability context K – 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 shows two examples of PCFGs. The example in Figure 3 (a) does not take context into consideration. The example in Figure 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.

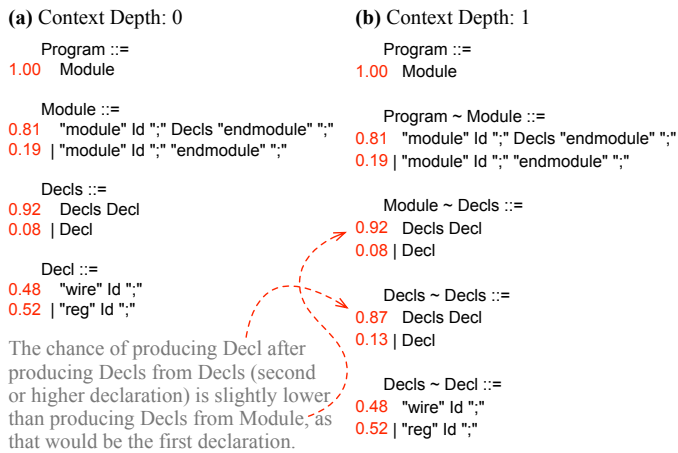


Fig. 3. Two probabilistic grammars. They recognize the same language, albeit with two different contexts of probabilities.

Each production rule is activated according to its proba-

bilities. The starting symbol of the grammar has probability 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, 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 eventually terminals, or the empty string, are produced. At the end of syntax generation, we obtain a skeleton of a Verilog design, as Example 2 illustrates.

Example 2. Figure 4 (a) shows an example of a design that is 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.

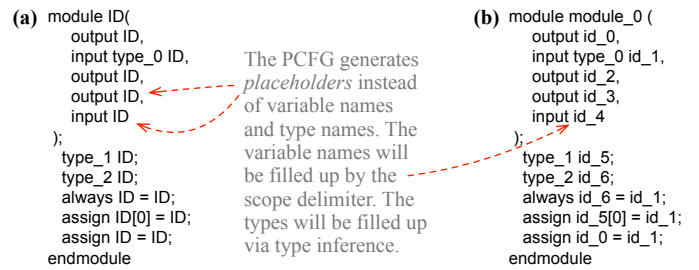


Fig. 4. Skeleton generation and symbol renaming.

B. Variable Renaming and Scope Creation

The design in Figure 4 (a) contains only placeholders where symbol names are expected. In the next state of 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 4 (b) shows the effect of renaming on the skeleton seen 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.

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 this scope can be created. Unlike noninstantiable namespaces, which provide a global or hierarchical organization of names (e.g., namespace in C++ or package in Java), instantiable namespaces allow for multiple copies, each maintaining its own state. Examples

include `struct` or `union` in C, and `class` in Python. Verilog provides one form of instantiable namespace in the `module` construct. SystemVerilog supports, additionally, interfaces, structs, unions, and classes. ChiGen is currently able to produce modules, unions, and structs (packed or unpacked). The presence of instantiable namespaces has an impact on the implementation of the scope delimiter, which must keep a table with all the namespaces instantiated in the current scope. Example 4 illustrates this feature.

Example 4. Figure 5 (a) shows a synthetic design with references to names defined within a module. Figure 5 (b) shows a design with names defined within a SystemVerilog *struct*. Both constructions are supported by ChiGen.

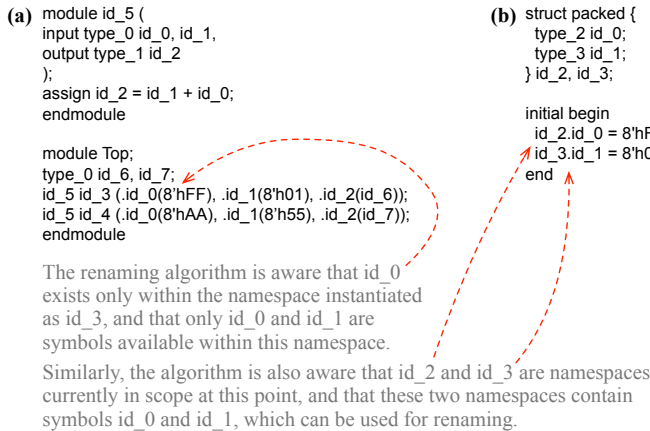


Fig. 5. Example of synthetic designs with instantiable namespaces.

C. Type Inference via Unification

The scope delimiter in Section III-B 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 [21]: 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 4). Example 5 summarizes this process, while the rest of this section provides details on each one of its two phases.

Example 5. Figure 6 (a) shows the seven pairs of constraints generated for the design in Figure 4 (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 the same. The result of unifying all the pairs appears in Figure 6 (b), where the type placeholders

have been replaced with actual type names in this updated version of our running example.

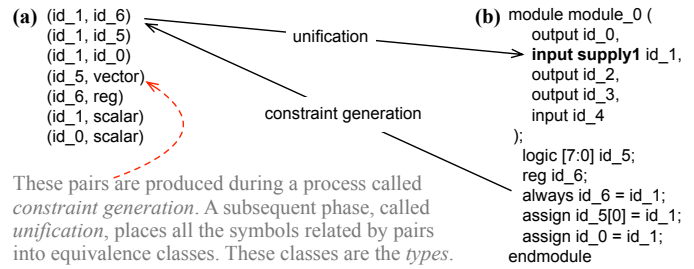


Fig. 6. Hindley-Milner Type Inference.

Constraint Generation. The process of constraint generation in Hindley-Milner type inference follows a visitor-like traversal of the abstract syntax tree (AST). As the visitor encounters each node, it introduces a set of type variables representing the types of identifiers and expressions at that node. Additionally, it generates a set of constraints, each of which is a pair expressing a type equivalence or compatibility requirement between two symbols—these symbols can be either type variables or user-defined identifiers. The constraints ensure that operations receive operands of appropriate types and that results are correctly propagated throughout the AST. Example 6 explains this step.

Example 6. When visiting an abstract syntax tree (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_{id_6} for `id_6`, t_{id_1} for `id_1`, and t_{id_2} for `id_2`
- 2) Create three constraints to associate each identifier with its type variable: (id_6, t_{id_6}) , (id_1, t_{id_1}) , (id_2, t_{id_2})
- 3) Create a constraint to enforce operand type compatibility: (t_{id_1}, t_{id_2})
- 4) Define two conditional constraints for result type inference: $(t_{id_1}, bitvector(N)) \Rightarrow (t_{id_6}, bitvector(N + 1))$, $(t_{id_2}, bitvector(N)) \Rightarrow (t_{id_6}, bitvector(N + 1))$

These ensure that:

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

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 7 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 7. Continuing with Example 6, 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 Section V, the chosen grammar yields a success rate of 50% with a probabilistic context of length one and 75% with a probabilistic context of length three.

D. 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* [14] in 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 8 shows the last two forms of injections. Example 9, at the end of this section, illustrates the first.

Example 8. Figure 7 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 8 describes our algorithm that implements module injection (injection of function calls and hierarchical references

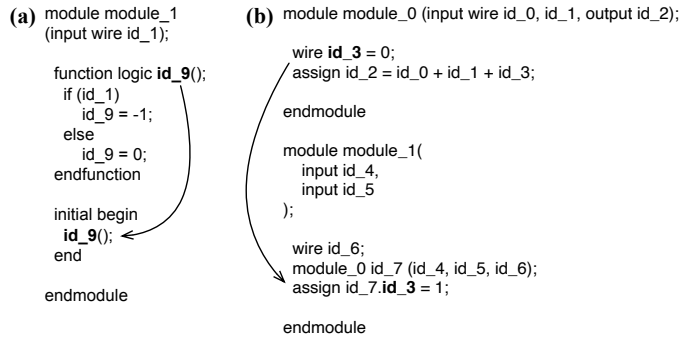


Fig. 7. (a) Injection of function call. (b) Injection of hierarchical reference.

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 8, 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.

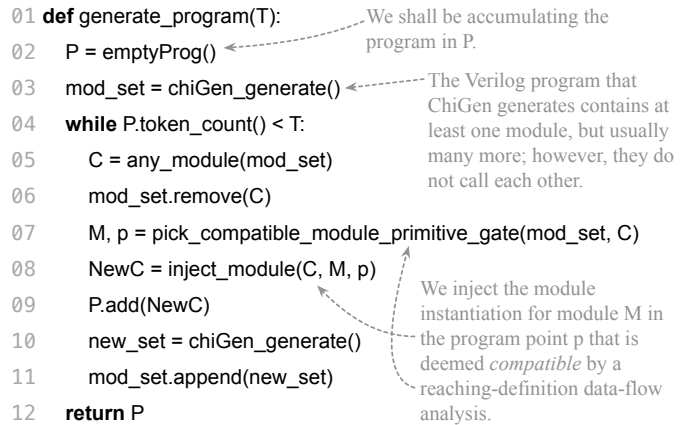


Fig. 8. The algorithm that implements module injection: it injects an instantiation of .

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 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 8. 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 Section III-A. 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 9. Figure 9 shows how the reaching-definition analysis enables code injection. Variables `id_3` and `id_4` reach Line 11 in Figure 9 (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 at Line 11 of `module_0`.

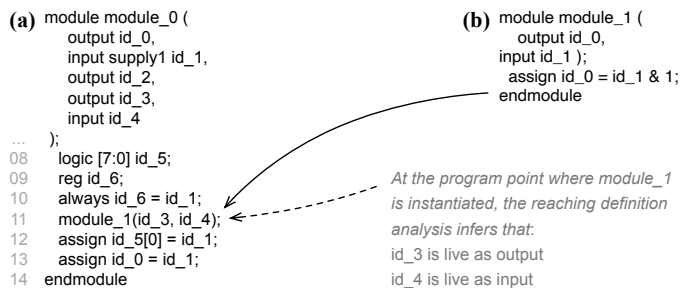


Fig. 9. Example of a design after module injection.

IV. CHIBENCH: THE TRAINING SET

The probabilistic grammar used in the open-source distribution of ChiGen (see Section III-A) 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 section details the methodology used to construct the ChiBench collection.

In order to build ChiBench, we have mined designs from open-source GitHub repositories, using GitHub’s REST API. We use GitHub’s API to build a list of candidate Verilog repositories. This 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.

A. 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 10 illustrates one such situation.

Example 10. The design in Figure 10, which specifies an 8-bit counter, will be filtered out by the syntactic filter. It contains a missing semicolon at Line 7. Such syntactically invalid files might occur in the mining process, as the repositories contain, for instance, files that are still under development.

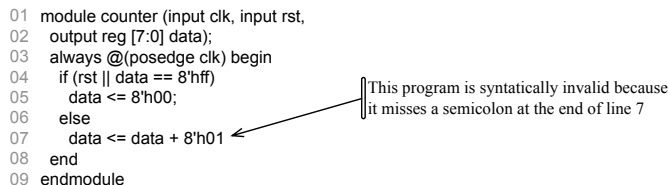


Fig. 10. Specification filtered out by syntactic verification.

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 restricted 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 11 better explains the semantic analysis’s role.

Example 11. Figure 11 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, wires cannot be assigned inside an `always` block; only `reg` or `logic` types can be written values in procedural contexts.

V. EVALUATION

This section evaluates the following research questions:

- RQ1: How diverse are the designs generated by ChiGen?
- RQ2: How does the coverage enabled by ChiGen-generated designs compare with that of real-world Verilog code when evaluated using typical EDA tools?

```

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

```

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

Fig. 11. Verilog specification that fails the semantic test.

- RQ3: What kinds of bugs can be uncovered through ChiGen-enabled fuzzing?
- RQ4: What is ChiGen’s throughput, measured in Verilog designs generated per second?
- RQ5: How has ChiGen’s effectiveness evolved across its releases since 2024?
- RQ6: What is the impact of the techniques described in Section III on the percentage and size of valid Verilog designs that ChiGen produces?

a) *Baselines*: We have used 10,000 ChiBench designs (Section IV) to train ChiGen’s probabilistic grammar. We compare ChiBench with Verismith [8], TransFuzz [20] and VlogHammer [27], which are other fuzzer collections. VlogHammer always generates the same 3,000 designs.

A. RQ1 – Diversity

This section evaluates the *syntactical diversity* of the designs generated by ChiGen. In Section V-B, we will indirectly assess their semantic diversity by examining the coverage they enable when used as input to EDA tools. We quantify syntactical diversity using two independent metrics, each measured over distinct populations of 1,000 files. Each population is generated by a different fuzzer. The metrics considered in this section are:

- 1) **Productions**: The number of unique production rules from the Verible grammar that are exercised when parsing the input files.
- 2) **Tokens**: The number of unique tokens from the Verible alphabet that are required to scan the input files.

We compare different fuzzers, and present, for perspective, the metrics found in the 10,000 programs used to train the ChiGen grammar (as seen in Section IV), plus the total number of productions and tokens in the Verible grammar. The latter represents the maximum number of productions and tokens that could be theoretically achieved. However, in practice, some of these grammar rules are not exercised in the ChiBench collection; hence, they cannot be produced by ChiGen, as they are assigned zero probability.

a) *Discussion – Unique Production Rules*: Figure 12 reports syntactical diversity in terms of the number of production rules exercised. 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 increases, so does the number of unique production rules exercised, approaching the diversity found in ChiBench, our ground truth. In contrast, VlogHammer, Verismith, and TransFuzz exercise substantially 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 mildly 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.

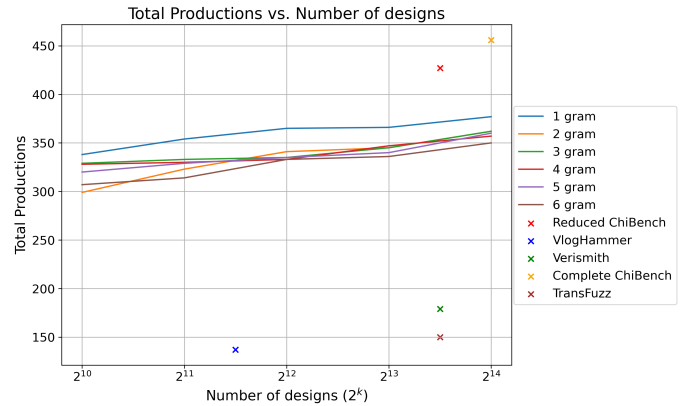


Fig. 12. Syntactical diversity of ChiGen designs, measured as the number of unique production rules in the Verilog grammar exercised when parsing a population of generated files.

b) *Discussion – Unique Tokens*: Figure 13 shows the number of unique tokens found in different populations of designs produced by ChiGen. The size of the probabilistic context affects these results only slightly, with a context of size one (1-gram) yielding the largest diversity. This trend also appears in Figure 12, and it is expected: the shorter the context, the greater the chance that different production rules (and consequently different tokens) will be activated, as rule selection is not constrained by previous activations. As also observed in Figure 12, Verismith and VlogHammer generate very few tokens (fewer than 60), whereas the different variants of ChiGen, except the 3-gram configuration, produce more than 200 unique tokens.

The relation between the results seen in Figures 12 and 13 is natural: Because ChiGen exercises more production rules than the other fuzzers, the programs it produces tend to be more diverse in terms of quantity and type of tokens. To demonstrate this fact, Figure 14 shows the number of 4-grams in a population of 10,000 designs. A 4-gram is a sequence of four kinds of tokens, e.g., `"ID → less_than → int → semi_colon"`. The six populations produced by ChiGen, with six different probabilistic contexts, approach the diversity observed in ChiBench, which is formed by actual Verilog codes. In contrast, the populations produced by the other fuzzers contain a very small number of different 4-grams.

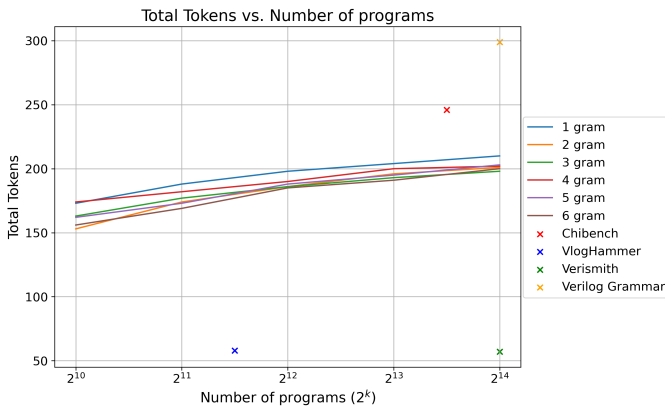


Fig. 13. Syntactical diversity of ChiGen designs, measured as the number of unique tokens when scanning a population of generated files.

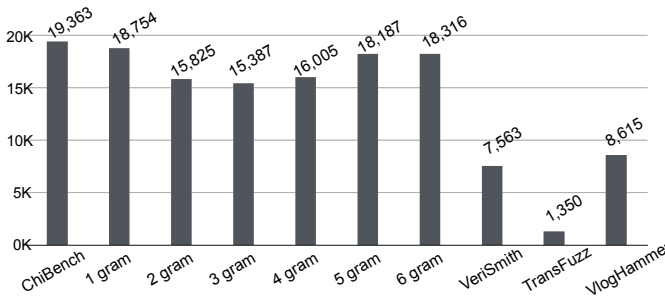


Fig. 14. Diversity of ChiGen designs, measured as the number of unique 4-grams: the number of different sequences formed by four kinds of tokens.

B. 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 `switch`, 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, meaning 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).

a) Discussion: In each of the eight charts in Figure 15, 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 with 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 V-A, which suggest that ChiGen designs are more 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.

C. RQ3 – Bugs

ChiGen was designed as a qualification tool for the Jasper Formal Verification Platform and has been used for this purpose within Cadence’s 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 I lists some of these issues our group is aware of.

To complement Table I, 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) [19]. Each campaign consists on 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.

a) Discussion: Table II 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,

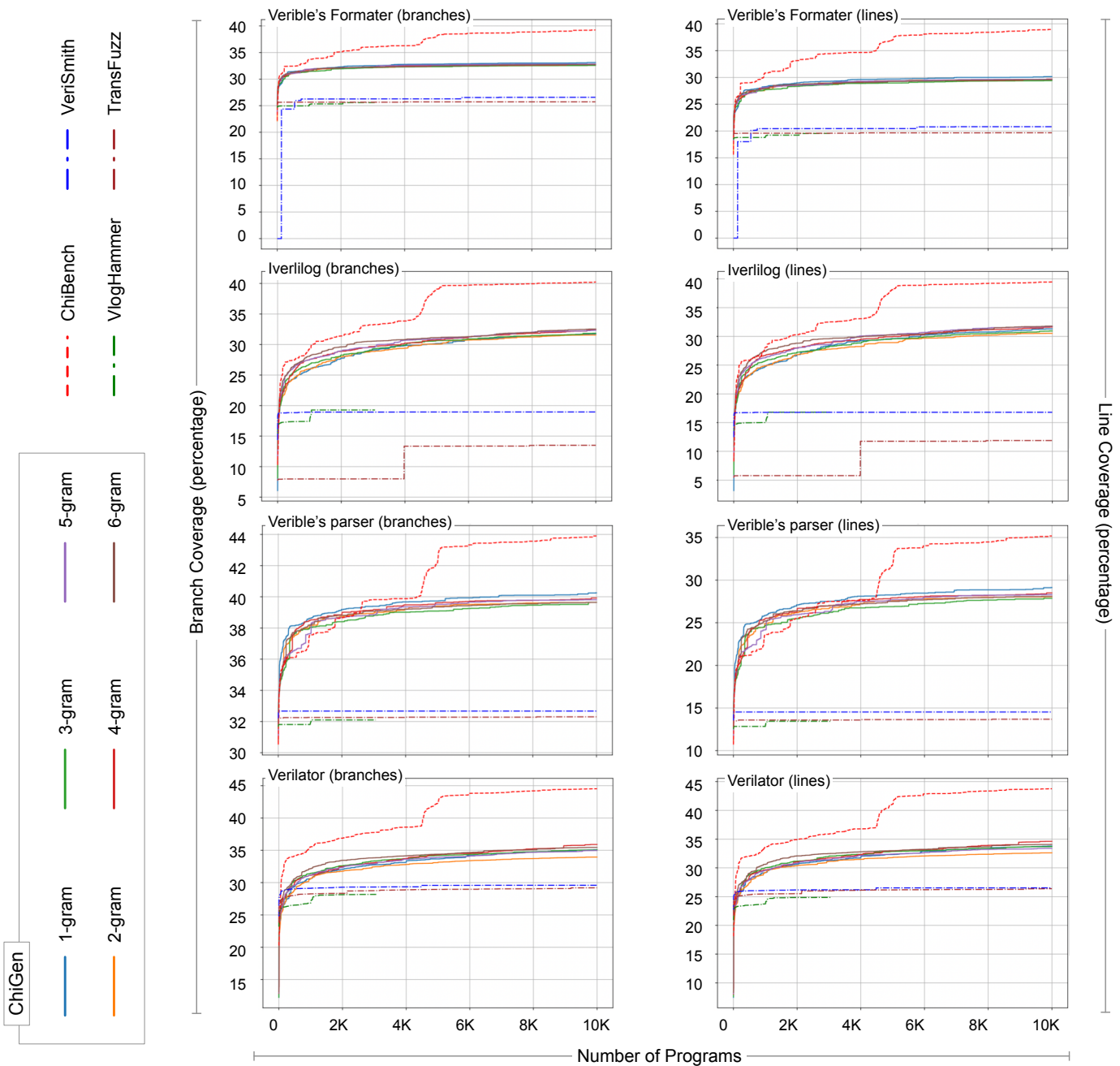


Fig. 15. Branch coverage obtained by testing Verible's syntactic analyzer with different sets of designs.

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.

D. RQ4 – Throughput

Throughput refers to the rate at which ChiGen produces test cases over a specific period of time. The higher its throughput, the more designs ChiGen produces. This section reports ChiGen's throughput 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.

a) *Discussion:* Figure 16 illustrates the throughput 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

TABLE I
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 a unary operation ($- -1$).
2364	Verible's Code Formatter	Fails to parse parameter declaration without qualifier ($\#(id_23=1)$).

TABLE II
CRASHES OBSERVED WITH 3,000 DESIGNS. 'ASAN/!ASAN': TOOL COMPILED WITH/WITHOUT ADDRESSSANITIZER; CB: CHIBENCH; VS: VERISMITH; VH: VLOGHAMMER; CG: CHIGEN; TF: TRANSFUZZ.

		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

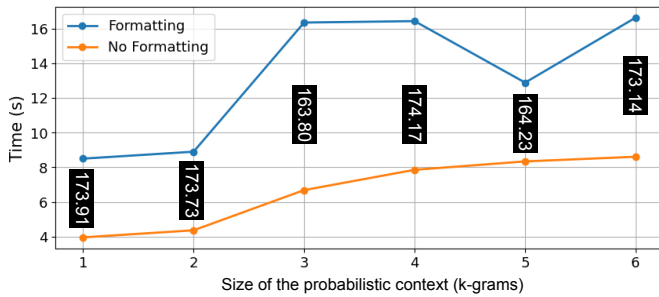


Fig. 16. Time to generate 100 designs, setting the lower limit of tokens to 150. Numbers in boxes show the average number of tokens.

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 on 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.

E. RQ5 – Software Evolution

Throughout its development, ChiGen was gradually augmented with different capabilities, some of which Section III discusses. When ChiGen was announced as an open-source tool in October 2024, it was at version 0.16; however, since v0.09, we have been tracking its effectiveness in generating valid Verilog designs. That release of the tool introduced

the type inference engine described in Section III-C. This section provides a glimpse of the evolution of this tool along three metrics: the percentage of valid designs it produces; the number of tokens (with repetition) present in these valid designs, and the average size of valid designs.

a) *Discussion*:: Figure 17 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 (Section III-D) in v0.12; and (iii) Addition of an extension for varying production probabilities in v0.16. This last extension, not detailed in Section III, 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 18 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% percent 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).

F. RQ6 – Ablation Study

Measuring the effectiveness of each technique discussed in Section III in isolation is not straightforward because, except for code injection, these techniques cannot be individually disabled. In other words, in releases after v0.09, it is no longer possible to run ChiGen without, for example, type

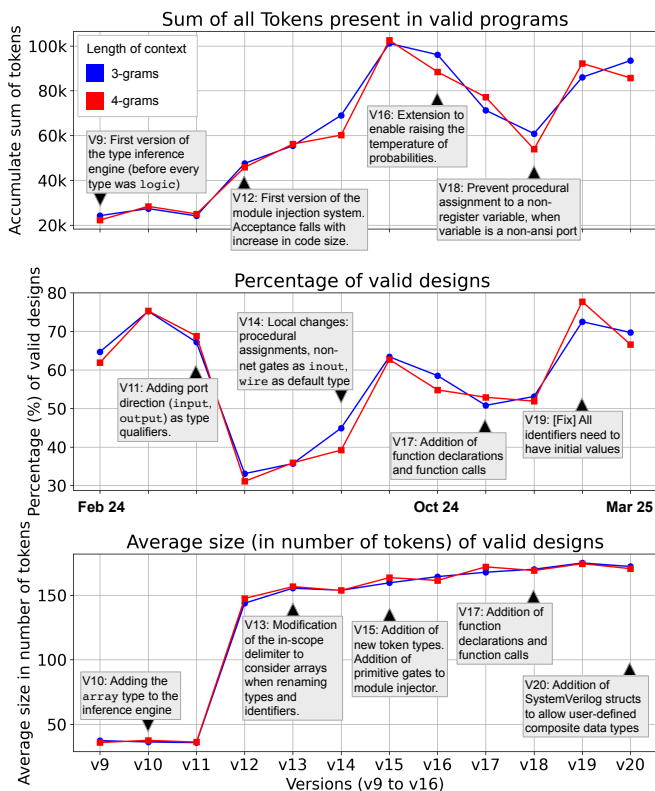


Fig. 17. The evolution of ChiGen, until its announcement as an open-source tool in October 2024. Each dot refers to 1,000 designs. Starting in v0.12, module injection uses a lower limit of 100 tokens.

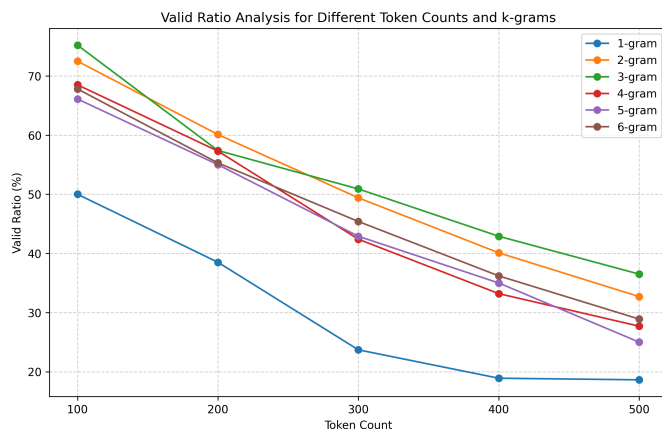


Fig. 18. Variation of the percentage of valid designs with the lower bound on the number of tokens in the designs produced by ChiGen.

inference. Nevertheless, we can approximate the contribution of each technique by evaluating past releases of ChiGen. This section extends Section V-E with such an ablation study. However, before discussing the results, we emphasize that the findings presented here are only an approximation. New releases incorporate not only additional techniques but also bug fixes and, at times, support for new syntactic constructs such as SystemVerilog features, which go beyond the algorithmic

mechanisms described in Section III.

a) *Discussion.*: Table III approximates the effectiveness of the techniques introduced in Section III, showing the percentage of valid designs among the 1,000 samples. To generate these results, we selected four past releases of ChiGen and used each to produce 1,000 Verilog designs with a probabilistic context of size two (2-gram). The analysis begins with the **Baseline** configuration, which includes only the Probabilistic Grammar (Section III-A). Subsequent rows show the cumulative benefit of incrementally adding the Scope Delimiter (Section III-B), Type Inference (Section III-C), and finally Code Injection (Section III-D).

TABLE III
ABLATION STUDY: EFFECTIVENESS OF CHIGEN'S CORE TECHNIQUES ON A POPULATION OF 1,000 SAMPLES GENERATED WITH 2-GRAMS.

Technique	Description	% Valid	Size (avg)	# Tokens
Baseline	Section III-A	22.2%	3.99	787
Scope Delimiter	Section III-B	23.2%	10.32	2,311
Type Inference	Section III-C	62.6%	10.53	6,418
Code Injection	Section III-D	74.5%	185.16	117,024

Table III also shows the average size of each valid design and the sum of all the tokens (counting repetitions) in all the valid designs. Collecting this data required one methodological adjustment for the Code Injection configuration. For the Baseline, Scope Delimiter, and Type Inference releases, design size is determined solely by the random activation of productions in the grammar trained as described in Section IV. With Code Injection, however, we impose a target design size by halting module insertion once the generated design reaches a threshold of 100 tokens. This constraint explains the substantial increase in the average design size (Size avg.) observed for the Code Injection configuration.

VI. RELATED WORK

RTL Fuzzing vs. Tool Fuzzing: ChiGen is a Verilog fuzzer, meaning that it generates designs to test EDA tools such as compilers, synthesizers, and formal verifiers. This approach differs from hardware fuzzers that focus on generating *input stimuli* (test vectors) to find bugs in hardware designs (i.e., the RTL itself). Hardware fuzzers include tools such as DifuzzRTL [10], TheHuzz [11], and RFuzz [13]. These tools are designed for Register-Transfer Level (RTL) verification. They take a *fixed* hardware design and generate sequences of input stimuli (test vectors) to reach deep architectural states or uncover functional bugs. This methodology, often described as “fuzzing hardware like software” [25], focuses on coverage-guided input generation for a specific Design Under Test (DUT). In contrast, ChiGen performs design-level fuzzing: it generates *Verilog designs*, rather than input stimuli.

Verilog Fuzzers: There are other fuzzers of verilog designs available as open-source tools [8], [20], [27]. In contrast to ChiGen, these tools gradually expand a core set of Verilog syntax, ensuring that each expansion results in a valid design. During the development of ChiGen, we had the opportunity to

engage with the authors of Verismith. We believe the following statement, shared in personal communication, highlights key differences between the two approaches: “*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.* [7]”

Grammar-Guided Fuzzing: Several tools can generate random strings based on the production rules of a context-free grammar [9], [4], [28]. In fact, the core ideas underlying grammar-guided fuzzing date back to the 1970s [17]. A well-known limitation of this approach is that, for programming languages, it is generally difficult to guarantee that the fuzzer will produce compilable programs. While a grammar-based fuzzer may generate syntactically valid programs, ensuring successful compilation typically requires constraints that go beyond the expressive power of context-free grammars. This limitation aligns with our experience developing ChiGen, and the data in Table III reinforces this point. When relying solely on syntactic fuzzing, we obtain fewer than 800 valid tokens across a population of 1,000 Verilog designs. In contrast, by applying the techniques described in Section III, we can increase this number to well over 100,000 tokens.

LLMs: Over the last years, the LLM revolution has brought to light many language models for Verilog [6], [15], [24], [26]. ChiGen is not a language model; it is a fuzzer, meaning that it does not attempt to shape the code toward 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.

Benchmarks: ChiGen can be used to generate Verilog benchmarks; however, it is not a benchmark collection. There already exist collections of benchmarks formed by hardware specification languages [1], [3], [5], [12], [16], [22]. In contrast to ChiGen, these collections are *immutable*: these previous works do not generate new benchmarks and incorporate them automatically into the database of available codes.

VII. CONCLUSION

This paper introduced 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, 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 Section II, 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.* [18]”.

ACKNOWLEDGMENT

This project was supported by a grant from Cadence Design Systems. The authors also acknowledge the support of FAPEMIG (Grant APQ-00440-23), CNPq (Grant #444127/2024-0), and CAPES (PRINT). The authors acknowledge the use of generative AI tools (Gemini 3.1 and GPT-4o) to improve the clarity and readability of the manuscript. Usage was solely for language revision; all technical content, ideas, and conclusions are the responsibility of the authors.

REFERENCES

- [1] Luca Amaru, Pierre-Emmanuel Gaillardon, Eleonora Testa, and Giovanni De Micheli. The epl combinational benchmark suite, February 2019.
- [2] IEEE Standards Association et al. Ieee standard for verilog hardware description language (ieee 1364-2005). <http://standards.ieee.org/>, 2006.
- [3] J. Babb, M. Frank, V. Lee, E. Waingold, R. Barua, M. Taylor, J. Kim, S. Devabhaktuni, and A. Agarwal. The raw benchmark suite: computation structures for general purpose computing. In *FCCM*, page 134, USA, 1997. IEEE Computer Society.
- [4] Bachir Bendrissou, Cristian Cadar, and Alastair F. Donaldson. Grammar mutation for testing input parsers. *ACM Trans. Softw. Eng. Methodol.*, 34(4), April 2025.
- [5] Franc Brglez, David Bryan, and Krzysztof Kozminski. Combinational profiles of sequential benchmark circuits. In *ISCAS*, pages 1929–1934, New York, USA, 1989. IEEE.
- [6] 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.
- [7] Yann Herklotz. Personal communication regarding the design of verismith, received on november 13th, 2024.
- [8] 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.
- [9] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. Grammarinator: a grammar-based open source fuzzer. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation, A-TEST 2018*, page 45–48, New York, NY, USA, 2018. Association for Computing Machinery.
- [10] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. DifuzzRTL: Differential fuzz testing to find cpu bugs. In *Security and Privacy (SP)*, pages 1286–1303, New York, USA, 2021. IEEE.
- [11] Rahul Kande, Addison Crump, Garrett Persyn, Patrick Jauernig, Ahmad-Reza Sadeghi, Aakash Tyagi, and Jeyavijayan Rajendran. TheHuzz: Instruction fuzzing of processors using Golden-Reference models for finding Software-Exploitable vulnerabilities. In *USENIX*, pages 3219–3236, Boston, MA, August 2022. USENIX Association.
- [12] Krzysztof Koźmiński. Benchmarks for layout synthesis—evolution and current status. In *DAC*, page 265–270, New York, NY, USA, 1991. Association for Computing Machinery.
- [13] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. RFuzz: coverage-directed fuzz testing of rtl in fpgas. In *ICCAD*, New York, NY, USA, 2018. Association for Computing Machinery.

- [14] Shaohua Li, Theodoros Theodoridis, and Zhendong Su. Boosting compiler testing by injecting real-world code. *Proc. ACM Program. Lang.*, 8(PLDI), June 2024.
- [15] Mingjie Liu, Nathaniel Pinckney, Brucec Khailany, and Haoxing Ren. Verilogeval: Evaluating large language models for verilog code generation, 2023.
- [16] 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.
- [17] Paul Purdom. A sentence generator for testing parsers. *BIT*, 12(3):366–375, September 1972.
- [18] Cary R. Comment on Icarus (IVerilog) issue tracker, posted on october 11th, 2024.
- [19] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: a fast address sanity checker. In *USENIX*, page 28, USA, 2012. USENIX Association.
- [20] Flavien Solt and Kaveh Razavi. Lost in Translation: Enabling Confused Deputy Attacks on EDA Software with TransFuzz. In *USENIX Security*, August 2025.
- [21] Martin Franz Sulzmann. *A general framework for Hindley/Milner type systems with constraints*. Yale University, New Heaven, USA, 2000.
- [22] 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.
- [23] Ari Takanen, Jared D. Demott, and Charles Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, Inc., USA, 2nd edition, 2018.
- [24] 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.
- [25] Timothy Trippel, Kang G. Shin, Alex Chernyakhovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks. Fuzzing hardware like software. In *USENIX*, pages 3237–3254, Boston, MA, August 2022. USENIX Association.
- [26] 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.
- [27] Claire Wolf. Vloghammer. <https://github.com/YosysHQ/VlogHammer>, 2021. Accessed: 2024-10-16.
- [28] José Antonio Zamudio Amaya. Shaping test inputs in grammar-based fuzzing. In *ISSTA*, page 1901–1905, New York, NY, USA, 2024. Association for Computing Machinery.



Rafael Sumitani received the B.Sc. degree in computer science from the Federal University of Minas Gerais (UFMG), Belo Horizonte, Brazil, where he is currently pursuing the M.Sc. degree. He is a Compiler Engineer at Cadence Design Systems, working on the design and implementation of the Xtena Neural Network Compiler (XNNC). Rafael developed the infrastructure used to collect ChiGen, the benchmark suite employed to train ChiGen, and contributed to ChiGen's early development, including the implementation of its probabilistic grammar.



Raissa Maciel received the B.Sc. degree in computer science from the Federal University of Minas Gerais. She is currently a Software Engineer at Cadence Design Systems, where she works on the development and testing of the Jasper[®] Formal Verification Platform. She implemented the infrastructure required to integrate ChiGen into Jasper's testing environment, enabling its effective use in production. Her experience provided valuable feedback that helped improve ChiGen's usability and its ability to generate non-trivial designs.



Augusto Mafra received the B.Sc. degree in electrical engineering from the Federal University of Minas Gerais (UFMG), where he is currently pursuing the M.Sc. degree. He is a Software Engineer at Cadence Design Systems, working since 2016 in the development for the HDL (Hardware Description Language) compiler of Jasper[®] Formal Verification Platform and has joined the ChiGen project with technical consulting on SystemVerilog and integration with Cadence tooling.



Mirlaine Crepalde received the M.Sc. degree in computer science from the Federal University of Minas Gerais (UFMG), Belo Horizonte, Brazil. She is currently a Software Engineering Manager at Cadence Design Systems, where she leads the development of the Jasper[®] Formal Verification Platform. She originally proposed the idea of developing a Verilog fuzzer to strengthen Jasper's testing methodology and has played a key role in fostering collaboration between Cadence Design Systems and UFMG.



Fernando Magno Quintão Pereira received the Ph.D. degree in computer science from the University of California, Los Angeles (UCLA). Since 2009, he is a professor in the Department of Computer Science at the Federal University of Minas Gerais, where he leads the Compilers Lab. In this project, he coordinated the development of ChiGen and defined the overall architecture of the fuzzing engine, including the integration of probabilistic grammars, type inference, and code injection.

João Victor Amorim Vieira received the M.Sc. degree in computer science from the Federal University of Minas Gerais in 2025, with the dissertation “Automatic Generation of Verilog Designs for Testing EDA Tools.” During his master's studies, João Victor worked on the development of ChiGen, contributing to major components of its initial codebase, including type inference and code injection. He is currently a Software Engineer at Cadence Design Systems, where he works on the Innovus Clock Design Team.

Luiza de Melo Gomes received the B.Sc. degree in computer science from the Federal University of Minas Gerais, and is currently a Software Engineer Intern at Google. As part of the ChiGen project, she worked as a research assistant at UFMG's Compilers Lab, contributing substantially to the implementation of the fuzzer. Her work was instrumental in achieving a compilation success rate that makes ChiGen effective in practice. Luiza has been admitted to the M.Sc. program in computer science at UFMG and is expected to begin her studies in 2026.

