Parse this! Summoning Context-Sensitive Inputs with Goblin

Robert Lorch
robert-lorch@uiowa.edu
The University of Iowa
Iowa City, Iowa, USA

Muhammad Daniyal Pirwani Dar
mdar@cs.stonybrook.edu
Stony Brook University
Stony Brook, New York, USA

Cesare Tinelli
cesare-tinelli@uiowa.edu
The University of Iowa
Iowa City, Iowa, USA

Omar Chowdhury
omar@cs.stonybrook.edu
Stony Brook University
Stony Brook, New York, USA

Abstract

Grammar-based fuzzers have shown immense promise in identifying bugs in software systems that have highly structured and intricate input formats (e.g., XML). Many of the current grammar-based fuzzers rely on context-free grammars (CFGs) to represent the target's input structure. CFGs, however, are often insufficient to precisely capture many application input formats that contain context-sensitive constraints. Application-specific fuzzers, albeit effective, lack generality necessary to make them adaptable to new applications. We present Goblin, a new input generation tool that helps bridge this gap. Given a context-free grammar annotated with semantic constraints written in Goblin's own DSL, Goblin generates inputs that both conform to the grammar and satisfy the constraints. A distinguishing feature of Goblin with respect to prior techniques targeting this problem is its support, through the use of an external SMT solver, for constraints expressed in arbitrary SMT theories. Goblin's input generation approach is itself inspired by DPLL-style SAT solvers and enjoys the following formal guarantees: *solution soundness* and *completeness*, and *refutation soundness*. In addition to comparing Goblin with prior work, we demonstrate its effectiveness by incorporating it into a grammar-based network protocol fuzzer.

1 Introduction

Real-world software systems, such as compilers, file parsers, and network protocol stacks, often have highly complex input specifications. Robust testing is difficult, and oversights can lead to unintended behavior or security vulnerabilities [23, 25–28]. Fuzzing is a common testing approach, but current work faces significant limitations when applied to systems with complex structured input. In blackbox generative settings [17, 34], one needs to capture the input specification without guidance from internal information from the system under test. Here, a common approach is *grammar-based fuzzing* [17, 34], where context-free grammars (CFGs) are used to capture the input structure. However, CFGs lack the expressiveness to specify *semantic constraints*, which require context sensitivity. For instance, a network message might contain a variable-length payload field f_1 and a *length* field f_2, with the requirement that the size of f_2's content, according to a suitable measure, must equal the value stored in f_1 (a *length constraint*). For many common file formats (e.g. CSV, XML, and more), performing context-free grammar-based fuzzing without taking these semantic constraints into account leads to the generation of a majority of inputs that do not pass early parsing stages of the system under test [31]. To address this problem, we present Goblin (**G**rammar-**o**riented **b**ranching with **l**ogical **i**nference and **n**-threading), a new context-sensitive input generation tool. Given a context-free grammar annotated with semantic constraints encoded in the Goblin DSL, Goblin generates inputs that conform to the grammar *and* satisfy the constraints. To our knowledge, Goblin is the first tool to efficiently generate semantically valid inputs for real-world protocols with rich constraints over a variety of data types. The richness of the constraint language is achieved through the use of off-the-shelf solvers for Satisfiability Modulo Theories (SMT) [4, 6].

Approach. In addition to being a general tool (in contrast to other, application-specific tools [7, 34]), Goblin demonstrates three core insights distinguishing it from prior work.

First, our approach conceptually *views context-free grammars as generators of abstract syntax trees (ASTs) rather than strings*. The AST view is achieved by encoding grammars as Algebraic Datatypes (ADTs) and ASTs as values of these ADTs and allows each AST leaf to be assigned a type—e.g., Int, BitVec(n), String. This enables a seamless integration of syntactic and semantic constraints thanks to use SMT solvers capable of reasoning over a combined theory of the types above. Thanks to the variety of theories supported by state-of-the-art SMT solvers, it also allows us to specify and natively perform constraint solving in various domains, such as bit vector constraints over network packets, that are out of reach for prior work [31]. Finally, this approach brings the benefits of static typing to the language, while prior work [31, 36] requires *unsafe casting* for non-string constraints.

Second, our approach provides a *minimal core language without sacrificing expressiveness*. Minimizing the set of *core language features* clarifies the formal foundations, simplifies the implementation, and avoids unnecessary special cases. In particular, it provides a solid foundation as an intermediate language for a future, more usable surface-level language which could be developed along the lines of well established and appreciated languages for attribute grammars or Constraint Logic Programming.

Third, our approach enables the expression of semantic constraints at the *production-rule level* rather than globally. This shift, which does not affect expressiveness, allows Goblin to integrate

naturally with fuzzing workflows involving *grammar-level* mutations, where grammar rules themselves are mutated rather than concrete inputs. This offers two distinct benefits: (*i*) mutated grammars function as a form of *root cause analysis*; and (*ii*) grammar-level mutations provide an extra mechanism for ensuring diversity of fuzzed inputs.

Goblin enjoys the formal guarantees of *solution soundness* (every AST generated by Goblin is a member of the language of the input grammar with constraints), *refutation soundness* (if Goblin reports Unsat, then the input grammar's language is empty), and *solution completeness* (if the input grammar's language is nonempty, then Goblin will find a member).

Evaluation and findings. We perform an experimental analysis with four case studies. Three of them (on XML, CSV, and Scriptsize C input generation) compare directly to prior work and demonstrate Goblin's ability to handle complex constraints with competitive performance. The fourth case study (WiFi SAE input generation) demonstrates Goblin's utility and ability to handle certain constraints and fuzzing workflows not handled by previous approaches.

Contributions. This paper makes the following contributions:

(*i*) We present a general approach for generating inputs that satisfy both context-free syntactic constraints and SMT-expressible semantic constraints.

(*ii*) We introduce Goblin, an input generator that supports a minimal yet expressive DSL for defining CFGs with production rule constraints over a rich set of data types.

(*iii*) We evaluate Goblin on four real-world case studies, showing that it scales to complex, constraint-rich input formats.

2 Motivation

We explain how Goblin integrates well with real-world fuzzers, and we discuss the relationship between Goblin and prior work.

2.1 Need for Local Constraints

Grammar-based fuzzers use the system-under-test's input language (represented as a grammar) as a guide for generating meaningful test inputs, especially when the test target has an intricate input format. There are two potential classes of grammar-based fuzzers: *Class I* and *Class II*. Class I fuzzers (e.g., [14, 15, 35]) generate a concrete input x from the input language $\mathcal{L}$ such that $x \in \mathcal{L}$. They then apply mutation operations on x to generate new test inputs. Class II fuzzers (e.g., [11, 20, 33]) directly mutate the original language $\mathcal{L}$ (i.e., the grammar production rules) to obtain a new language $\mathcal{L}'$. They then generate concrete test inputs y such that $y \in \mathcal{L}'$. The claimed advantage of Class II fuzzers is that the mutation operations can themselves be aware of the input structure. In addition, when a concrete input triggers a bug in the system under test, the mutated grammar that was used to generate the input can help with root cause analysis.

The problem addressed by Goblin is how to construct concrete inputs x from the language $\mathcal{L}$ generated by a given grammar $\mathcal{G}$. The grammars considered by Goblin go beyond context-free grammars as they can include constraints over grammar elements. Goblin supports a Domain-Specific Language (DSL) for expressing grammars that capture the input language $\mathcal{L}$ of a target system. The DSL allows the addition of constraints to each production rule of the grammar, which we refer to as *local, or production-rule-level, constraints*.

The features above allow Goblin to serve as an input generator for Class II fuzzers, similarly to others tools such as ISLa [31] and Fandango [36]. A natural question with such tools is: *If a fuzzer performs grammar production-level mutations, then what should happen to the semantic constraints?* In ISLa, the input is a *pair* ⟨grammar, constraints⟩, where the constraint language allows quantification, structural predicates over the generated term, and pattern matching over them. In some cases, it may be possible to mutate the constraints in analogy with the production rule mutations; however, it is not clear how it is done in general. In contrast, in Goblin, attaching constraints locally to production rules rather than the entire input grammar, makes constraint mutation much more natural. An informal example of production rules with local constraints (not in the Goblin DSL's actual syntax), is shown in Listing 1 where local constraints are provided in curly braces after each rule.

Listing 1: Grammar-level mutations

```
COMMIT ::= SEQ GROUP_ID SCALAR
  { GROUP_ID is equal to 13  and  SEQ is greater than 4 }
RG_CONT ::= RG_LENGTH RG_TY RG_LIST
  { RG_LENGTH is the length of RG_LIST }
...
⇝
COMMIT ::= SEQ RG_LIST SCALAR
  { SEQ is greater than 4 }
RG_CONT ::= RG_LENGTH RG_TY GROUP_ID
  { GROUP_ID is equal to 13 }
...
```

Listing 1 illustrates a *crossover mutation*, where the rules after the arrow are the result of a mutation to the rules before the arrow. In this mutation, two production rules are selected, and one nonterminal from the first rule (`GROUP_ID`) is swapped with one from the second rule (`RG_LIST`). It is easy to maintain the semantic constraints over `GROUP_ID` and `SEQ` as they can follow the corresponding nonterminals. The constraint over `RG_LIST`, on the other hand, is more difficult to adapt to the mutation as it involves a nonterminal that is moved by the mutation and one that is not. In our example, we simply drop that constraint in the mutated grammar. Despite this sort of difficulty, which can be addressed in a number of ways, we claim that local mutations bring more benefits as they can help boost the diversity of fuzzed inputs while maintaining the well-formedness of constraints.

2.2 Comparison with Prior Work

Goblin and ISLa. The most relevant prior work is the one embodied by ISLa [31], another tool for context-sensitive input generation. Goblin distinguishes itself from ISLa in several ways. First, the semantics of ISLa restrict it to support natively only string and integer constraints. This is a strong limitation because, for instance, bit vector constraints are common in network packet formats (see our working example in Listing 2). While one could technically encode bit vector constraints into string constraints, this is tedious, makes the input grammar less direct, and is also less performant at generation time. In general, extending ISLa to natively support other data types in grammars seems nontrivial. In contrast, Goblin

can readily be extended to any datatypes and constraints supported by its backend SMT solver. Another difference is that, while both tools perform backtracking search over the space of possible solutions to generate elements of their input grammar, Goblin's search algorithm employs chronological backtracking to leverage the incremental solving capabilities of SMT solvers. Finally, Goblin's core language is presently minimalistic when compared to ISLa's.

Goblin and Fandango. Fandango [36] also targets semantics-aware input generation, but uses search heuristics based on genetic algorithms instead of constraint solving. This choice is justified by its authors as leading to greater performance. We found however that, while Fandango is very efficient for many examples, its performance varies greatly depending on the input constraints. Fandango's genetic algorithm steers search towards semantically correct outputs through fitness functions favoring candidate outputs that are the *closest* to satisfying the semantic constraints. However, Fandango presently relies on a set of hard-coded fitness functions. In our experience, if none of thems happens to be a reliable distance metric for the user constraints, Fandango's performance degrades substantially. If none of the built-in fitness functions works for an input, Fandango struggles to produce even a single valid instance. Several simple examples demonstrate this phenomenon—for example, (*i*) an input grammar with two nonterminals `<nt_a>` and `<nt_b>` with a single semantic constraint `<nt_a> == <nt_b>` (one equality constraint), (*ii*) an input grammar with two nonterminals with a single semantic constraint `str(<nt_a>) == str(<nt_b>) + "foo"` (one string constraint), and (*iii*) an input grammar with an integer nonterminal and a single semantic constraint `int(<nt>) in {1, 2, 3}` (one set membership constraint). In addition, when we tried to use Fandango to synthesize a valid input for the WiFi SAE packet grammar and constraints that we explore as part of our experimental evaluation of Goblin (see Section 8), Fandango was not able to produce a single instance. The Fandango examples demonstrating our encodings for these problems is available at [22].

3 Goblin's Language and Semantics

We give an overview of the Goblin language and its semantics, starting with a working example which we will reference throughout the paper, to give an informal intuition for how Goblin works.

3.1 Working Example

Listing 2 contains a constrained grammar in Goblin's DSL syntax, describing the format of a simple communication network packet whose components consist of bit vector values. The syntax is intentionally reminiscent of that of parser generators like YACC which extend BNF-style production rules with additional information in braces, semantic constraints in our case. For increased readability we color nonterminals, like `<PACKET>`, in red and *symbolic terminals*, like `<TYPE>` (akin to *tokens* in parser generator languages) in teal.

Listing 2: Working example in Goblin DSL

```
1 <PACKET> ::= <TYPE> <AUX> <PAYLOAD>
2   { <AUX> <- <PAYLOAD>.<F1> bvmul <PAYLOAD>.<F2>;
3     <TYPE> = 0x01 => (<PAYLOAD>.<BYTES>.<BYTE> bvugt 0x20 and
4                       <PAYLOAD>.<BYTES>.<BYTE> bvult 0x7E); };
5 <PAYLOAD> ::= <F1> <F2> <BYTES> { <BYTES>.<OPT> bvugt 0x0; };
6 <BYTES> ::= <BYTE> <BYTES> | <BYTE> | <OPT>;
7 <TYPE> :: BitVec(8) { <TYPE> = 0x01 or <TYPE> = 0x02; };
8 <AUX> :: BitVec(8); <BYTE> :: BitVec(8) { <BYTE> bvult 0x88;};
9 <F1> :: BitVec(8); <F2> :: BitVec(8); <OPT> :: BitVec(4);
```

Line 1 describes a production rule for `<PACKET>`, a single option producing three *syntactic categories* `<TYPE>`, `<AUX>`, and `<PAYLOAD>`. Lines 2 through 5 attach to the production rule two (semicolon-separated) *semantic constraints*, that is, conjunctive constraints on the possible values of the nonterminal or symbolic terminal involved. The first states that `<AUX>` takes the bit vector resulting from the product of `<PAYLOAD>.<F1>` and `<PAYLOAD>.<F2>`, where `<PAYLOAD>.<F1>` (resp., `<F2>`) denotes the value of the `<F1>` (resp., `<F2>`) component of `<PAYLOAD>`, as per the second production rule. Observing that the nonterminal `<BYTES>` is defined as a sequence of values of `<BYTE>` (bit vectors of size 8), the second constraint of the `<PACKET>` rule imposes range limits on the first byte of `<PAYLOAD>` when `<TYPE>` has value `0x01`. Lines 8, 10, and 11 contain *type annotations* for the symbolic terminals in the grammar, which ascribe a type to their possible values. The concrete syntax of these values (e.g., `1`, `0x7E`) is built-in. Also built-in are the supported types (`Bool`, `BitVec(8)`, ...) and operators over them (`=`, `bvmul`, `or`, ...). As shown in lines 8 and 9, type annotations too can carry semantic constraints.

3.2 Language Features

We define the language syntax using a standard extension of Backus-Naur form [2] where the * operator denotes zero or more instances, the + operator denotes one or more instances, and the square brackets denote optional elements.

$$
\begin{aligned}
\langle \text{S} \rangle &::= \langle \text{element} \rangle^+ \\
\langle \text{element} \rangle &::= \langle \text{ty\_annot} \rangle \mid \langle \text{prod\_rule} \rangle \\
\langle \text{ty\_annot} \rangle &::= \langle \text{nt} \rangle \texttt{ : } \langle \text{type} \rangle \texttt{ [\{ } \langle \text{constraint} \rangle^+ \texttt{ \}]} \\
\langle \text{prod\_rule} \rangle &::= \langle \text{nt} \rangle \texttt{ ::= } \langle \text{nt} \rangle^+ \texttt{ [\{ } \langle \text{constraint} \rangle^+ \texttt{ \}]} \\
&\quad \texttt{[| } \langle \text{nt} \rangle^+ \texttt{ [\{ } \langle \text{constraint} \rangle^+ \texttt{ \}]]}^+\texttt{;} \\
\langle \text{constraint} \rangle &::= \langle \text{derived\_field} \rangle \texttt{; } \mid \langle \text{expr} \rangle \texttt{;} \\
\langle \text{derived\_field} \rangle &::= \langle \text{nt} \rangle \texttt{ <- } \langle \text{expr} \rangle \\
\langle \text{expr} \rangle &::= \langle \text{expr} \rangle \ \langle \text{binop} \rangle \ \langle \text{expr} \rangle \mid \langle \text{unop} \rangle \langle \text{expr} \rangle \mid \\
&\quad \langle \text{f} \rangle (\langle \text{expr} \rangle, \ \ldots, \ \langle \text{expr} \rangle) \mid \langle \text{p} \rangle (\langle \text{expr} \rangle, \ \ldots, \ \langle \text{expr} \rangle) \mid \\
&\quad \langle \text{nt\_expr} \rangle \mid \text{constant} \mid (\langle \text{expr} \rangle) \\
\langle \text{nt\_expr} \rangle &::= \langle \text{nt} \rangle \mid \langle \text{nt} \rangle \texttt{.} \langle \text{nt\_expr} \rangle \\
\langle \text{nt} \rangle &::= \texttt{< identifier >} \\
\langle \text{type} \rangle &::= \texttt{Bool} \mid \texttt{Int} \mid \texttt{String} \mid \texttt{Set}(\langle \text{type} \rangle) \ \ldots \\
\langle \text{f} \rangle &::= \texttt{length} \mid \texttt{bv\_cast} \mid \ldots \\
\langle \text{p} \rangle &::= \texttt{leq} \mid \texttt{geq} \mid \ldots \\
\langle \text{binop} \rangle &::= \texttt{and} \mid \texttt{or} \mid \ldots \\
\langle \text{unop} \rangle &::= \texttt{not} \mid \ldots
\end{aligned}
$$

In the BNF, we did not enumerate all possible types, predicate symbols, function symbols, and constants. In principle, we support every type, predicate symbol, and function symbol that has a direct translation to SMT-LIB [5] or cvc5's [3] support for non-standard theories [10]. (In practice, we have not yet implemented support

for all of the types, function symbols, and predicate symbols, but will glady extend the input language where demand arises.)

Semantic constraints. Each production rule is optionally annotated with a set of Boolean expressions, all of which must be satisfied by the output term at every instance in the derivation where the production rule is invoked. In principle, the expression language supports every type, predicate symbol, and function symbol that has a direct translation to theories specified in the SMT-LIB standard [5] or supported by the cvc5 solver [3, 10].[1]

Nonterminals vs. symbolic terminals. The *syntactic categories* of symbolic terminals and nonterminals are identified syntactically for having names delimited by angular brackets (< >). The two are distinguished by the fact that former can only have type annotations while the latter can only have production rules. We have *symbolic* terminals rather than *concrete* ones since in Goblin we wish to capture only *abstract syntax*, the logical structure of the input specification, omitting purely syntactic details. This naturally positions Goblin as an ADT term generator built for handling constraints over symbolic terminals with different types. Serialization is a later step of the pipeline and can be handled directly by Goblin or with a user-defined serialization function.

Refinement types. Types can also have associated semantic constraints, which provide a form of *type refinement*. Type constraints are similar to production rule constraints, but they are applied to type annotations rather than production rules. For example, one can use a refinement type to model the Internet Header Length (IHL) field of an IPv4 header [29] as `<IHL> :: BitVec(4) { int_to_bv(4, 4) bvult <IHL> and <IHL> bvult int_to_bv(4, 16); };`, where the refinement specifies the valid range of values for the field.

Dot notation. When writing local constraints, we use *dot notation* to constrain the values of syntactic categories that are descendants of the ones in the current production rule. For instance, in line 5 of Listing 2, we use the expression `<PAYLOAD>.<BYTES>.<BYTE>` to access the first byte of the payload. More concretely, this expression refers to `<BYTES>`'s child `<BYTE>`, where `<BYTES>` is the child of `<PAYLOAD>` in the derivation.

Derived fields. Some constraints are not amenable to analysis by SMT solvers due to being prohibitively expensive to reason about or prohibitively difficult to express in the SMT-LIB language [5]. We handle these constraints separately from the rest of the search problem through a language feature called *derived fields*, in which the syntax `<field> <- <expression>` denotes that the value for syntactic category `<field>` can be computed with expression `<expression>`. For example, in Listing 2, the `<AUX>` symbolic terminal is a derived field since its value can be calculated from other parts of the generated term. Without the derived field, we would have to pass this nonlinear constraint to the SMT solver, likely causing nontermination. Derived fields are often useful in practical scenarios, e.g., in (*i*) RSA-encrypted payloads (e.g., in TLS records), (*ii*) HMAC-SHA256 authentication tags (e.g., in IPsec packets), and (*iii*) CRC-32 checksums (e.g., in Ethernet frames). Since derived fields are computed outside the SMT solver, it is theoretically straightforward to support any computable function within derived field definitions.

[1]At the time of this writing, only a subset of these theories, their types and their operators are supported by Goblin but we plan to extend this set on demand.

3.3 Semantics

Ambiguous dot notation. Before formally defining the language semantics, we first informally discuss a tricky aspect of the language: *ambiguous dot notation references*. Consider Listing 3.

Listing 3: Ambiguous dot notation reference

```
1 <A> ::= <B> <B> <C> { <B>.<D> > <C>; };
2 <B> ::= <D> <D>;
3 <D> :: Int; <C> :: Int;
```

Above, the expression `<B>.<D>` intuitively could refer to either the first or second child `<D>` of either the first or second occurrence of `<B>`. Goblin treats all ambiguous references of this form as *implicitly universally quantified* over the structure of generated terms—that is, one can view the above constraint as internally desugaring to `<B>[0].<D>[0] > <C>[0]; <B>[0].<D>[1] > <C>[0]; <B>[1].<D>[0] > <C>[0]; <B>[1].<D>[1] > <C>[0]`, where the bracket notation $[i]$ of a syntactic category uniquely indicates which occurrence of the syntactic category is being referenced. Furthermore, consider Listing 4, where `<B>` gets a second production rule referencing `<D>`.

Listing 4: Ambiguous dot notation reference II

```
1 <A> ::= <B> <B> <C> { <B>.<D> > <C>; };
2 <B> ::= <D> <D> | <D>;
3 <C> :: Int; <D> :: Int;
```

At term generation time, if `<B>`'s second production rule is chosen, the constraint `<B>[0].<D>[1] > <C>[0]`, say, is dropped from consideration since `<B>[0]` does not have a child `<D>[1]` (instead, it has a single child, `<D>[2]`).

CFGs with constraints. To capture the language semantics, we first formally define an *augmented context-free grammar* (ACFG) G. Intuitively, G is a context-free grammar (CFG) where each production rule can carry additional (context-sensitive) constraints. Also, we replace terminal symbols with type annotations which specify that the given symbolic terminal ranges over the given type.

More concretely, G is a 4-tuple $(\mathbb{C}, R, \Gamma, S)$ where $\mathbb{C}$ is a finite set of syntactic categories; $R : \mathbb{C} \times \mathbb{C}^+ \times C^*$ is a relation capturing the production rules, with each production rule additionally carrying zero or more semantic constraints; Γ is a set of $\mathbb{C} \times \Lambda$ pairs denoting type annotations; and $S \in \mathbb{C}$ is the distinguished start symbol. Additionally, we require that each syntactic category either is mentioned on the left-hand side of one or more production rules, or has exactly one type annotation, but not both.

ASTs. Rather than working with raw strings, we define the semantics of an ACFG G as the set of valid abstract syntax trees producible by the grammar (denoted $\mathcal{L}_{\text{AST}}(G)$). We define an *abstract syntax tree* (AST) *tr* for a given ACFG G as a 3-tuple $(V, \vec{E}, \ell)$, where $(V, \vec{E})$ denotes a directed graph without (even undirected) cycles, and $\ell(.)$ is a labeling function that takes a vertex as input and returns its label. Here, a label can either be a syntactic category or a primitive value (e.g., the numeral `100` or the string literal `"foo"`). We abuse notation to allow ℓ to also return a syntactic category's associated integer index for disambiguation (as discussed in Section 3) where convenient. Also, get_children $:: AST \times V \rightarrow V^*$ takes the expected semantics, and we abuse notation by using $\in$ to mean both set and list membership. Additionally, *tr* has two syntactic well-formedness requirements:

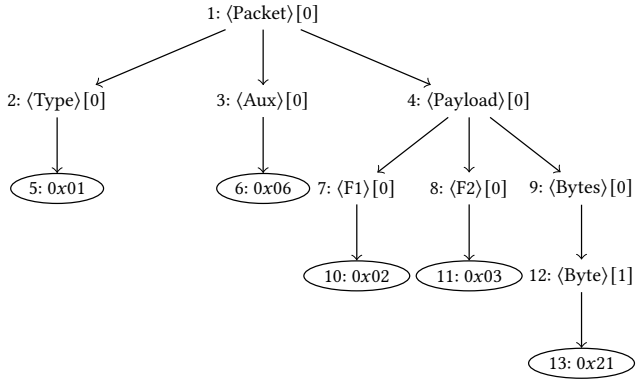

Figure 1: Goblin AST

Vertices are identified by indices on the left-hand side of each colon, and the labels are to the right. Leaf nodes are circled. Occurrences of syntactic categories are explicitly disambiguated with an index in square brackets.

(*i*) $\ell(\text{root}(tr)) = S$ (the AST is rooted at the start symbol);
(*ii*) for every $v \in V$, either
  (a) for every $v_1, \ldots, v_n \in \text{get\_children}(tr, v)$, $(\ell(v), (\ell(v_1), \ldots, \ell(v_n)), \_) \in R$, or
  (b) there is only a single $v_1 \in \text{get\_children}(tr, v)$ and $(\ell(v), \tau) \in \Gamma$ (for some τ), or
  (c) v is a leaf vertex with $\ell(v) \in \tau$, with τ the type of $\ell(\text{parent}(v))$ ($\ell(\text{parent}(v))$ must be a symbolic terminal).

Informally, each node in the tree is either a non-leaf node with children representing an application of some production rule in G, a non-leaf node with a single child representing some type annotation in G, or a well-typed leaf.

An example AST is depicted in Figure 1. We can verify that it is indeed a valid AST for the working example: It satisfies condition (*i*), as it is rooted at <PACKET>, which is the start symbol. For condition (*ii*) we can verify that each vertex in the tree expands to a set of children capturing some production rule (e.g., vertex 1 with label <PACKET> has children with labels <TYPE>, <AUX>, and <PAYLOAD>, corresponding to the first production rule) or type annotation (e.g., vertex 2 with label <TYPE> has a single child with label 0*x*01, which aligns with the given type `BitVec(8)`).

Interpretation function. Next, we define the meanings of terms in a production rule constraint for a given AST *tr* by defining an interpretation function $\mathcal{I}_{tr}(t)$, which outputs the denotation of term t in AST *tr*. Additionally, we overload $\mathcal{I}_{tr}$ to also map function and predicate symbols to their fixed semantic interpretations. Now notice that in our definition of $\mathcal{I}_{tr}$, (*i*) the variable assignment is determined by the structure and labels of *tr* rather than by a mapping of variable names to values, (*ii*) terms evaluate to a distinguished symbol $\top$ if they reference syntactic categories that are not present in the given tree, and (*iii*) we assume all terms are well-typed (e.g., primitive values take the expected types, each function symbol has a given set of input types and an output type, and each function symbol is passed the correct number of arguments with the correct types). For primitive values (e.g., integers), the denotation is defined by the identity function. Otherwise, $\mathcal{I}_{tr}$ is defined according to the first applicable case below.

$\mathcal{I}_{tr}(\langle \text{sc} \rangle[i]) = \top$
  if $(\ell(\text{root}(tr)), \_, \_) \in R$ and
  $\{v \in \text{get\_children}(tr, \text{root}(tr)) \mid \ell(v) = \langle \text{sc} \rangle[i]\} = \emptyset$
$\mathcal{I}_{tr}(\langle \text{sc} \rangle[i]) = \mathcal{I}_{tr'}(\langle \text{sc} \rangle[i])$ for tr' rooted at the only
  $v \in \text{get\_children}(tr, \text{root}(tr))$ such that $\ell(v) = \langle \text{sc} \rangle[i]$
  if $(\ell(\text{root}(tr)), \_, \_) \in R$
$\mathcal{I}_{tr}(\langle \text{sc} \rangle[i]) = \ell(v)$ for the unique $v \in \text{get\_children}(tr, \text{root}(tr))$
  if $(sc, \_) \in \Gamma$
$\mathcal{I}_{tr}(\langle \text{sc} \rangle[i].\langle \text{sc\_expr} \rangle) = \top$
  if $\{v \in \text{get\_children}(tr, \text{root}(tr)) \mid \ell(v) = \langle \text{sc} \rangle[i]\} = \emptyset$
$\mathcal{I}_{tr}(\langle \text{sc} \rangle[i].\langle \text{sc\_expr} \rangle) = \mathcal{I}_{tr'}(\langle \text{sc\_expr} \rangle)$ for tr' rooted at the only
  $v \in \text{get\_children}(tr, \text{root}(tr))$ such that $\ell(v) = \langle \text{sc} \rangle[i]$
$\mathcal{I}_{tr}(f(t_1, \ldots, t_n)) = \top$ if some $\mathcal{I}_{tr}(t_i) = \top$
$\mathcal{I}_{tr}(f(t_1, \ldots, t_n)) = \mathcal{I}_{tr}(f)(\mathcal{I}_{tr}(t_1), \ldots, \mathcal{I}_{tr}(t_n))$

Now consider AST *tr* rooted at vertex 1 (from Figure 1); we will compute $\mathcal{I}_{tr}$(<PAYLOAD>.<F1> `bvmul` <PAYLOAD>.<F2>) (each index `[0]` is omitted for brevity). This is equal to the fixed interpretation of `bvmul` applied to $\mathcal{I}_{tr}$(<PAYLOAD>.<F1>) and $\mathcal{I}_{tr}$(<PAYLOAD>.<F2>) (last case above). Now $\mathcal{I}_{tr}$(<PAYLOAD>.<F1>) = $\mathcal{I}_{tr'}$(<F1>) = $\mathcal{I}_{tr''}$(<F1>), where *tr*′ is rooted at vertex 4 and *tr*″ is rooted at vertex 7. These equalities follow by the fifth and second cases above, respectively. Finally, $\mathcal{I}_{tr''}$(<F1>)= 0*x*02 by the third case above, as vertex 7 has a single child representing a well-typed leaf with label 0*x*02. By similar logic, $\mathcal{I}_{tr}$(<PAYLOAD>.<F2>)= 0*x*03. Multiplying 0*x*02 and 0*x*03 gives the final result: $\mathcal{I}_{tr}$(<PAYLOAD>.<F1> `bvmul` <PAYLOAD>.<F2>) = 0*x*06.

Satisfaction relation. We define a satisfaction relation $\models_{\mathcal{G}}$ that captures whether or not a given constraint in G is satisfied by a given abstract syntax tree. The satisfaction relation uses $\top$ analogously to the term semantics.

$tr \models_{\mathcal{G}} \varphi$ if $\mathcal{I}_{tr}(t_i) = \top$ for some subterm t_i of φ; otherwise,
$tr \models_{\mathcal{G}} p(t_1, \ldots, t_n)$ if $(\mathcal{I}_{tr}(t_1), \ldots, \mathcal{I}_{tr}(t_n)) \in \mathcal{I}_{tr}(p)$
$tr \models_{\mathcal{G}} \neg\varphi$ if $tr \not\models_{\mathcal{G}} \varphi$
$tr \models_{\mathcal{G}} \varphi_1 \wedge \varphi_2$ if $tr \models_{\mathcal{G}} \varphi_1$ and $tr \models_{\mathcal{G}} \varphi_2$
$tr \models_{\mathcal{G}} \varphi_1 \vee \varphi_2$ if $tr \models_{\mathcal{G}} \varphi_1$ or $tr \models_{\mathcal{G}} \varphi_2$
$tr \models_{\mathcal{G}} \varphi_1 \Rightarrow \varphi_2$ if $tr \not\models_{\mathcal{G}} \varphi_1$ or $tr \models_{\mathcal{G}} \varphi_2$

We can observe that the AST in Figure 1 satisfies the constraint <AUX> <- <PAYLOAD>.<F1> `bvmul` <PAYLOAD>.<F2>. First, we compute $\mathcal{I}_{tr}$(<PAYLOAD>.<F1> `bvmul` <PAYLOAD>.<F2>) and $\mathcal{I}_{tr}$(<AUX>), both giving value 0*x*06. Since <- is a predicate symbol with a fixed interpretation that both inputs are equal to each other, the constraint is satisfied by the second case of the satisfaction relation.

Semantics of a Goblin input. Below, get_subtrees(·, ·) is a function that takes as input an AST t and a syntactic category ⟨sc⟩, and returns all subtrees in t rooted at vertices v with $\ell(v) = \langle \text{sc} \rangle$. Finally, resolve :: $C \to \mathcal{P}(C)$ performs the desugaring of ambiguous dot notation references discussed previously, and we lift $\models_{\mathcal{G}}$ to sets of constraints as expected. Put together, the semantics of an input, $[\![G]\!]$, is defined as the set of ASTs in the language of G that also satisfy all the semantic constraints.

$$[\![G]\!] = \{t \mid t \in \mathcal{L}_{\text{AST}}(G) \wedge \forall(\text{nt}, \_, \text{constraints}) \in R. \\ \forall s \in \text{get\_subtrees}(t, \text{nt}).\ \forall \varphi \in \text{constraints}.\ s \models_{\mathcal{G}} \text{resolve}(\varphi)\}$$

4 Design Overview

Next, we give an informal overview of Goblin's input generation process. The main pipeline is depicted in Figure 2.

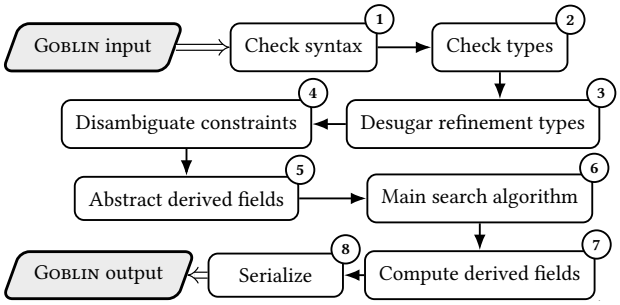

Figure 2: Main pipeline

Context-free well-formedness. At the syntax checking step ①, we perform a generalization of the standard context-free language emptiness check [18]. The first aspect of this check is indeed language emptiness: if the language is empty, we can quickly report the problem to the user (likely from a modeling error) before launching into the main algorithm. At the context-free level, language emptiness stems from non-well-founded recursion (i.e., recursion without a base case), leading to infinite derivations. Intuitively, the algorithm proceeds by iteratively expanding a set of nonterminals that are known to be able to produce terminal strings until reaching a fixpoint, and then checking if the set contains the start symbol. The generalization comes from performing a more conservative check which mandates that *all* reachable nonterminals must be a member of this set, as violation still likely signals a modeling error.

Listing 5: Syntax checks

```
<SAE_PACKET> ::= <COMMIT> | <CONFIRM>;
<COMMIT> ::= <FIELD> <RG_ID_LIST>;
<RG_ID_LIST> ::= <RG_ID> <RG_ID_LIST>;
<CONFIRM> ::= <FIELD1> <FIELD2>;
...
```

An example input rejected by our syntax checker (inspired by real experience with our WiFi case study) is in Listing 5. While the language of the grammar is non-empty through a derivation involving `<CONFIRM>`, the `<COMMIT>` nonterminal cannot derive a finite string due to a missing base case in the definition of `<RG_ID_LIST>`. Hence, the input is conservatively rejected, as no finite derivation through `<COMMIT>` is possible.

Other syntax checks. Other syntax checks include the detection of dangling identifiers (e.g., a semantic constraint `<sc> > 0` with no type annotation or production rule defining `<sc>`) and invalid dot notation references (e.g., an expression `<nt>.<st>` when `<nt>` has no instance of `<st>` in any of its production rules).

Type checking. Goblin requires type annotations for symbolic terminals and has very limited support for polymorphism. Therefore, type checking is straightforward as every expression's type can be inferred bottom-up.

Refinement type constraints. We support refinement types by desugaring type annotation constraints to production rule constraints at step ③ by essentially inlining the type annotation constraint for some symbolic terminal `<st>` at every production rule option containing `<st>` (on the right-hand side). Inlining simplifies the search algorithm (and its formalization), as we do not have to consider a separate case for type annotations with semantic constraints. We demonstrate the inlining with a simple example in Listing 6.

Listing 6: Refinement types

```
<S> ::= <NAT> <NAT>;
<NAT> :: Int { <NAT> >= 0; };
⇝*
<S> ::= <NAT>[0] <NAT>[1]
  { <NAT>[0] >= 0; <NAT>[1] >= 0; };
<NAT> :: Int;
```

Disambiguating dot notation. Notice that in Listing 6, after inlining, the constraint `<NAT> >= 0` applies to a production rule with two instances of the `<NAT>` symbolic terminal. Under the hood, the constraint is disambiguated in step ④ to two constraints `<NAT>[0] >= 0` and `<NAT>[1] >= 0`, taking the semantics of *implicit universal quantification*. The disambiguation makes it explicit which instances of each syntactic category are referenced by a constraint.

Derived fields. To make derived fields computable outside the SMT solver, we reject two scenarios: (*i*) we disallow mutual (cyclic) dependencies between derived fields, and (*ii*) we disallow including derived fields in (non-derived) semantic constraints. To detect cyclic dependencies, we borrow an efficient technique of circularity detection in attribute grammars [9, 37]. In step ⑤, we build a directed graph for each production rule. There is a node for each syntactic category on the right-hand side of the production rule and an edge from `<sc1>` to `<sc2>` when `<sc1>` is a derived field defined in terms of `<sc2>`. If this graph contains a cycle, then the input is rejected for a cyclic dependency. This algorithm is a fast overapproximation that is computable in polynomial time, but works for "every practical example" [9]. To detect when derived fields are mentioned in (non-derived) semantic constraints, we simply scan every semantic constraint for dot notation expressions mentioning syntactic categories that are associated with derived fields. After these checks, these derived fields are removed from the grammar and replaced with opaque, "stub" symbols that record the positions of the derived fields within the grammar.

Listing 7: Derived fields

```
<S> ::= <TYPE> <AUX> <PAYLOAD>
  { <AUX> <- <Payload>.<F1> bvmul <Payload>.<F2>; };
...
⇝
<S> ::= <TYPE> dep_leaf <PAYLOAD>;
...
```

Listing 7 demonstrates how derived fields like `<AUX>` are abstracted away—they are replaced by special symbolic leaf values `dep_leaf` within the input grammar. The `dep_leaf` value will still be present in the term generated by the main search algorithm in step ⑥, say, `(S (Type 0b00000002) (AUX dep_leaf) (Payload (F1 0x00) (F2 0x00) (Bytes (Byte 0x00))))`. In step ⑦, we compute the value of `dep_leaf` to give output term `(S (Type 0b00000002) (AUX 0b00000000) (Payload (F1 0x00) (F2 0x00) (Bytes (Byte 0x00))))`.

Main solving algorithm. The main search algorithm, step ⑥, is described in detail in Section 5. Intuitively, the solving algorithm iteratively builds a candidate solution by walking through

the grammar and choosing production rules to expand. The algorithm explores the search in an *iterative deepening* matter to control the size of candidate solutions (larger terms lead to more expensive constraint solving) and to avoid exploring rabbit holes in the search space too deeply. When semantic constraints are encountered, they are eagerly asserted to an SMT solver in its incremental mode [5, 6] to quickly detect unsatisfiable states. When unsatisfiable states are detected, the search backtracks to the most recent decision (i.e., most recent choice of production rule to expand) if possible. Otherwise, the search will report that the input problem is unsatisfiable or continue the search with an increased depth limit.

Serialization. Goblin produces ADT terms rather than strings. More specifically, Goblin produces Lisp-style terms, where each term is either (*i*) a concrete leaf-level value (say, `1000` or `"foo"`), (*ii*) a symbolic leaf-level value for derived fields (say, `dep_leaf`), or (*iii*) a constructor string paired with a list of subterms (say, `(S (LEN 0b00000000) (TYPE 0b00000000) (PAYLOAD []))`). To produce strings, the user defines a serialization function mapping these Lisp-style terms to strings outside of Goblin. Optionally, Goblin provides a few built-in serialization functions in step ⑧ geared towards network packet generation that can be invoked with a command-line argument. Because network packet formats can mix fields of different endianness, Goblin supports (optional) symbolic terminal *endianness annotations* to allow the user to specify the endianness of the serialization of each symbolic terminal.

5 Goblin's Search Algorithm

Here, we describe various aspects of Goblin's constraint solving approach, referencing our working example from Listing 2.

Derivation trees. Before delving into the search algorithm, we introduce the concept of *derivation trees* (formally defined in [22]), which are abstract syntax trees that are allowed to have syntactic category labels at leaf vertices. Example derivation trees are given in Figure 3—note that some of the tree leaves correspond to unexpanded syntactic categories. We call a derivation tree *closed* if none of the leaves correspond to syntactic categories (i.e., if it is an AST). Thus, derivation trees intuitively represent (potentially unfinished) candidate solutions. The graphical representation of derivation trees is verbose, so in the remaining text we will refer to derivation trees by the concatenation of all the leaf nodes.

5.1 Context-Free Input Generation

We begin with a simplified discussion of a naive approach for input generation in the context-free setting (i.e., no semantic constraints). We will walk through iterative refinements of this naive approach to eventually arrive at the Goblin algorithm.

Context-free naive approach. A naive but sound approach is to randomly walk through the grammar and expand syntactic categories until there are none left to expand. Such an expansion is pictured in Figure 3, where the first derivation tree steps to the second derivation tree as the `<BYTES>` nonterminal is expanded according to its first production rule.

Context-free normalization. When constructing a candidate solution, some decisions are forced—namely, production rule expansions for nonterminals with only one production rule option, and symbolic terminal expansions. To trim down the search space, we normalize the derivation tree by greedily performing forced expansions. Then, when backtracking, we always backtrack to a normalized derivation tree. Normalization is similar to unit propagation in DPLL [12]—before making new decisions, we propagate consequences forced by previous decisions. In the working example, the search starts with derivation tree `<PACKET>`. However, since `<PACKET>` only has one expansion option, it takes a normalization step to `<TYPE> <AUX> <PAYLOAD>`. Here, we can perform more normalization—for example, `<TYPE>` and `<AUX>` are symbolic terminals and only have one expansion option. Fully normalizing, we start the search with the first derivation tree in Figure 3. Upon backtracking, we will backtrack only to the normalized tree (and never to just `<PACKET>`).

Making decisions. When a derivation tree is normalized but *open* (i.e., not closed), there must exist some open leaf with multiple expansion options. At this point, we make a *decision* by selecting an expansion option to try. Whenever we make a decision to expand derivation tree *dt*, we push *dt* to a global decision stack, depicted in Figure 3. Then, if we ever want to backtrack to the most recent decision, we simply pop the stack. On the stack, we also track each tree's set of visited expansions so we do not explore the same expansions multiple times.

Context-free iterative deepening. For recurive grammars, termination is not guaranteed, so one needs a mechanism to prevent endlessly exploring cycles in the grammar. We address this with an *iterative deepening search* that tracks the current search depth (incremented at each expansion) and backtracks when the current depth limit is exceeded. For example, in Figure 3, if the second derivation tree's search depth exceeds the current depth limit, we will backtrack to the last decision made, which is the first derivation tree. At this point, we will try another expansion. In this case, `<BYTES>` can be expanded to `<BYTE>` through the second production rule, which immediately normalizes to a closed derivation tree. If no closed derivation trees can be found at the current depth limit, the search algorithm restarts with an increased depth limit.

5.2 Context-Sensitive Input Generation

We move past the context-free setting and now consider augmenting the approach to handle context-sensitive semantic constraints.

Semantic constraints. At a high level, we can summarize constraint solving with four points.

(1) Whenever a decision is made, push an assertion level (with `(push 1)`) to the incremental SMT engine.
(2) Whenever an expansion or normalization step is performed, assert all the local semantic constraints (using `(assert)`) for the corresponding production rule, and check their feasibility (with `(check-sat)`).
(3) Whenever a decision is backtracked, pop an assertion level from the incremental SMT engine.
(4) Do *not* retrieve a model (using `(get-model)`) until the derivation tree is closed.

Notice in Figure 3, the constraint on the `<TYPE>` field is at the first assertion level since it was encountered during the initial normalization steps. However, the constraint on `<BYTE>` was not added until assertion level two, since `<BYTE>` was not encountered until the expansion to `dt2`. With these principles, we notice two desirable

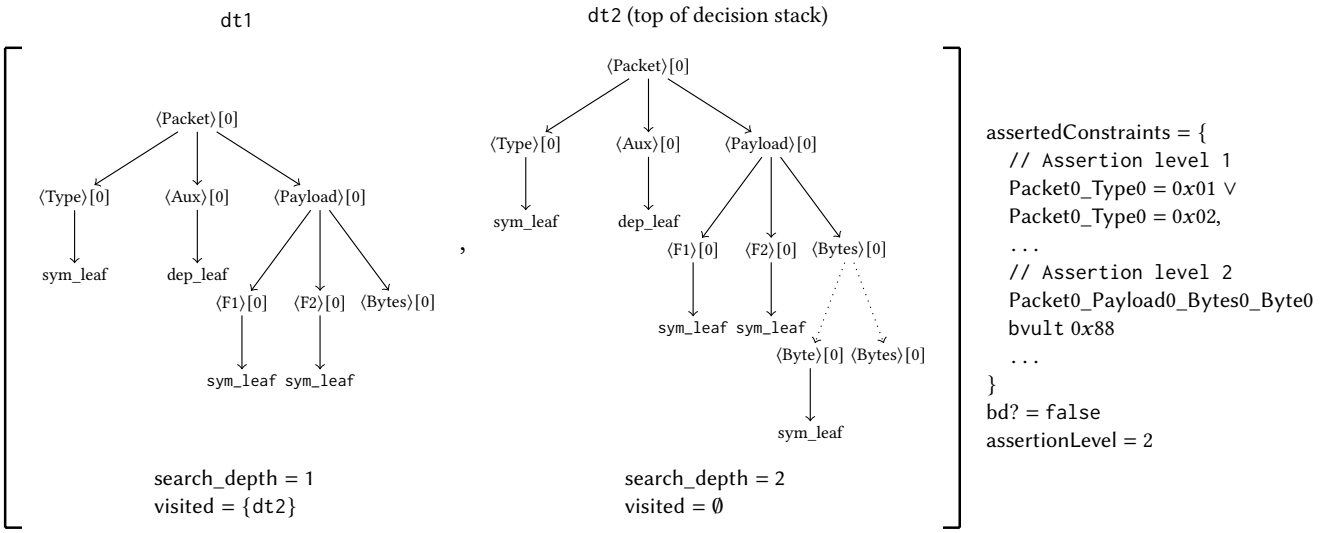

Figure 3: Goblin algorithm snapshot

Snapshot of the main elements of global state in the Goblin search algorithm. The two derivation trees represent the decision stack with the top on the right. Local (per tree) state is below each derivation tree, while global state is to the right of the stack. Edges originating from normalization steps (forced expansions) are solid, while edges originating from decisions are dashed.

outcomes: (*i*) The approach quickly detects when a candidate solution becomes infeasible (because the SMT engine will report Unsat at point 2); (*ii*) we can efficiently mend candidate solutions (within the solver's internal state) which are promising but imperfect by delaying the call to `(get-model)` (point 4).

Goblin and SMT semantics. Goblin's semantics (described earlier in Section 4) are defined in terms of a satisfaction relation $\models_{\mathcal{G}}$ between abstract syntax trees and Goblin formulas, while SMT semantics are defined in terms of a satisfaction relation $\models_{\mathcal{T}}$ between first-order models and first-order logic formulas, modulo theories of interest. To help bridge this gap in semantics, we define a function universalize_c which translates Goblin constraints to SMT constraints. Intuitively, the universalization function replaces dot notation expressions with SMT symbolic constants, where the constant name is produced by serializing the path from the root node to the corresponding leaf node in the derivation tree. In Figure 3, a universalized (and disambiguated) instance of the Goblin constraint `<BYTE> bvult 0x88` is `Packet0_Payload0_Bytes0_Byte0 bvult 0x88`.

Inapplicable constraints. Point (2) from the **Semantic Constraints** paragraph was an oversimplification. Notice that the constraint set in Figure 3 does not include the constraint `<BYTES>.<OPT> bvugt 0x0`—this is because the constraint is considered *inapplicable*. More precisely, a constraint is inapplicable in a given derivation tree *dt* if it includes some dot notation subexpression that references a syntactic category which is not present in *dt* after expansion and normalization. In this case, `<BYTES>` does not have a child `<OPT>` in the second derivation tree of Figure 3, so the constraint is not asserted. In fact, the constraint is considered trivially satisfied according to Goblin's semantics (Section 3). If a constraint is inapplicable, we store it in a constraint set C to possibly assert later, since an inapplicable constraint can become applicable with future expansions. Hence, at each expansion we must inspect the new local constraints as well as those in C. When backtracking, we can then remove the constraints associated with the backtracked expansions from C.

5.3 Algorithm description

We discuss the full algorithm at a high level of abstraction. More details (pseudocode, formalization, and source code) are available at [22]. We frame the algorithm in a more general setting that supports the generation of multiple context-sensitive inputs and also supports the ability to conclude Unsat for input grammars with empty languages.

Multiple solutions. After finding a solution single solution, Goblin (*i*) pops to the zeroth assertion level, (*ii*) asserts a blocking clause which prevents the same solution from being found multiple times, and (*iii*) pushes an assertion frame to start the next search from the first assertion level. With this strategy, Goblin ensures that blocking clauses are never popped from the assertion stack.

Unsatisfiable inputs. Goblin derives Unsat when the current constraint set is unsatisfiable, there is no decision to backtrack, and no candidate solutions have been discarded due to hitting the depth limit. If the depth limit has caused backtracking since the last restart, we cannot soundly conclude Unsat since there may be solutions we artificially discarded due to the depth limit.

Core logical steps. The search algorithm is phrased in terms of 11 core logical steps, distinguished with an all-caps serif font (e.g., Decide). These logical steps can be viewed as pairs of preconditions and rewrite rules, where the rewrite rule updates the global state and the preconditions on the global state denote when each rule can be applied. Decide expands the candidate solution DT at some open leaf associated with a nonterminal symbol carrying more than one production rule option (intuitively capturing a new decision). Propagate is the same as Decide, but it applies when there is only

one unvisited expansion left. NormalizePR and NormalizeTA perform normalization steps on *DT*. Assert asserts a constraint from the current constraint set at the current assertion level to the incremental SMT solver. More concretely, we assert grammar production rule constraints encountered at some application of Decide, Propagate, or NormalizePR which also constrain part of the current *DT*. BacktrackDepth reverts the last decision and pops an assertion level from the solver when the depth limit has been exceeded. BacktrackUnsat reverts the last decision and pops an assertion level from the solver when the set of asserted constraints has become unsatisfiable. RestartDepth and RestartUnsat are analogous to BacktrackDepth and BacktrackUnsat, except they apply when there is no prior decision to backtrack. Instead, there is a full restart, and the depth limit is increased. Solve instantiates a closed derivation tree with concrete values, assuming the set of asserted constraints is satisfiable. Fail gives up when the current set of asserted constraints is unsatisfiable, there is no prior decision to backtrack, and we have not backtracked due to the depth limit since the last restart.

Global state. Most notably, the global state contains (i) the current candidate term under construction *DT*, (ii) the current search depth d, (iii) the current search depth limit L, (iv) the current assertion level assertionLevel, (v) the current set of relevant SMT constraints C, and (vi) a Boolean flag bd? which tracks whether we have backtracked due to hitting the depth limit since the last restart. Note that we also need to manage the state of the incremental SMT solver (i.e., its current assertion level and set of active assertions), as well as information about visited expansions so we do not return to the same state after backtracking. Formal, detailed descriptions of how the 10 core logical steps modify the global state is given in Section 6.

Algorithm Pseudocode. See Algorithm 1 for the algorithm pseudocode. We begin in line 1 by initializing the global state and then loop (line 2) until we have either found a suitable number of solutions or have determined that there are no more. While the current candidate term is open (line 3), we expand it with Decide (line 5) or Propagate (line 7), depending on the number of unvisited expansions, and then normalize (lines 9 and 10). If this new expansion pushed beyond the current search depth limit (lines 11), we backtrack if possible and otherwise restart (incrementing the depth limit). Otherwise, we are still within the depth limit (line 16), and we assert all collected applicable SMT constraints from past expansions (line 17). If we reached an unsatisfiable state due to these assertions (line 19), we either fail, backtrack, or restart, depending on whether we have previous decisions to revert and whether or not we have previously backtracked due to hitting the depth limit. If the state is still satisfiable, we keep looping (line 3). At some point, we may complete a candidate term such that there are no more open leaves (line 26). At this point, we check for unsatisfiable states as in line 16. If we have a closed candidate term in a satisfiable state, then we can retrieve a model from the SMT solver (line 29) and instantiate it into the candidate term, generating a solution. If we want multiple solutions, we can retrieve multiple models and push blocking clauses (line 29), and eventually restart for diversity reasons (line 30) before continuing the top-level loop.

Algorithm 1 High-level Search Algorithm

```
1:  initializeGlobalState()
2:  while you still want more solutions do
3:    while ¬allLeavesClosed(dt) do
4:      if there is more than one unvisited expansion then
5:        Decide
6:      else
7:        Propagate
8:      while ¬is_normalized(dt) do
9:        NormalizePr  (if applicable)
10:       NormalizeTa  (if applicable)
11:     if current search depth  d > depth limit  L then
12:       if assertionLevel = 1 then
13:         RestartDepth
14:       else
15:         BacktrackDepth
16:     else
17:       for all c ∈ constraint set C do
18:         Assert  if  c  is applicable
19:       if smt_check_sat() = UNSAT then
20:         if assertionLevel = 1 ∧ ¬bd? then
21:           Fail
22:         else if assertionLevel = 1 then
23:           RestartUnsat
24:         else
25:           BacktrackUnsat
26:   if smt_check_sat() = UNSAT then
27:     {Same logic as the code block starting on line 19}
28:     continue
29:   iteratively Solve and push blocking clauses
30:   Restart  for diversity of solutions
```

6 Formal Approach

In this section, we present a formal description of the approach described in Section 4 by formulating it as a calculus. We present some preliminaries, give an intuitive description of the calculus, and then prove that the calculus offers certain guarantees, namely solution soundness, refutation soundness, and solution completeness.

Derivation trees. We define a *derivation tree* as an abstract syntax tree that is potentially unfinished. More precisely, the labeling function ℓ is updated to have type $V \to N \cup (\bigcup \Lambda) \cup \{\text{None}\}$; case (c) of syntactic restriction (ii) is updated to allow $\ell(v) = \text{None}$; and we introduce a new case (d) of syntactic restriction (ii) which allows v to be a leaf vertex with $\ell(v) \in N$. We denote the language of derivation trees of a grammar G as $\mathcal{L}_{\text{DT}}(G)$, which is analogous to $\mathcal{L}_{\text{AST}}(G)$ but for derivation trees rather than ASTs.

Configurations. The calculus operates on **configurations**, which are either eight-tuples $(DT, \boldsymbol{O}, C, A, DS, \text{visited}, \text{depth}, L)$ or the distinguished symbol $\perp$, denoting that there is no solution. Each rule of the calculus updates the current configuration to a new configuration, called a **conclusion**, given that the specified preconditions hold. In each rule, unprimed variables denote the current configuration, and the primed variables represent the updated configuration. If the primed version of some configuration variable is left

undefined in the conclusion of a rule, it is assumed that the configuration variable is left unchanged. A rule **applies** to a configuration C if all the rule's preconditions hold for C. A configuration is considered **saturated** if it is not $\bot$ and no rule applies. The **initial configuration** for a given ACFG $G = (N, R, \Gamma, S)$ is the eight-tuple $(DT, \mathcal{O}, \{\}, \{\}, [], \{\}, \{\}, L)$, where DT is the derivation containing only a single node labeled with S, and $\mathcal{O}$ is at the zeroth assertion level with no assertions pushed, and L is some natural number (initial depth limit).

Proof trees. A **proof tree** is a directed tree where every node maps to a configuration, and all the children of each node are obtained by the application of some applicable rule to the node. A proof tree T' **derives** from a proof tree T if T' is obtainable by applying a rule to one of T's leaves. A **derivation** is a finite or countably infinite sequence of proof trees such that each proof tree in the sequence derives from the previous proof tree and every proof tree has an initial configuration at the root. Then, a **refutation** is a finite derivation where all leaves of the final derivation tree are $\bot$, and a **solution** is a finite derivation where all leaves of the final derivation are saturated.

Note that since each rule in the calculus only produces a single resulting configuration, every proof tree is a path with a single leaf.

Configuration description. The configuration variables are now described more intuitively.

(1) DT is a derivation tree that represents the (potentially unfinished and backtrackable) construction of an output term.
(2) $\mathcal{O}$ represents an incremental SMT oracle, supporting operations push, pop, assert, and check_sat.
(3) C represents a set of relevant SMT constraints.
(4) A represents the set of constraints that have been asserted since the last pop() of the incremental SMT oracle.
(5) DS represents a decision stack of derivation trees, supporting operations push, pop, and is_empty.
(6) visited $:: \mathcal{DT} \times V \to \mathcal{P}(R)$ represents a map of $\langle$ derivation tree, vertex $\rangle$ pairs to the set of production rule expansions that have been explored.
(7) depth $:: \mathcal{DT} \to \mathbb{N}$ represents a map from a derivation tree to its search depth.
(8) $L :: \mathbb{N}$ represents the current depth limit.

A configuration is **satisfiable** if DT, or some derivation tree in DS, can be extended to an AST in $\mathcal{L}_{\text{AST}}(G)$, and otherwise, it is **unsatisfiable**. Every AST in $\mathcal{L}_{\text{AST}}(G)$ that is an extension of DT or some derivation tree in DS, only using expansions not captured by visited, is called a **model** of the configuration.

Analogously, an ACFG G is called **satisfiable** if its language is nonempty, and otherwise it is **unsatisfiable**.

Informal rule explanations. See Section 5

Other preliminaries. The set of **active assertions** for $\mathcal{O}$ is defined as the set of assertions at the current, or lower, assertion levels of $\mathcal{O}$. A constraint φ is considered **applicable** in a derivation tree dt if for all terms t in φ, $\mathcal{I}_{dt}(t) \neq \top$ (that is, every dot notation expression references children that are actually present in the derivation tree). We use $\mathcal{M}$ to denote a model returned by check_sat; open_leaves $:: \mathcal{DT} \to \mathcal{P}(V)$ returns the set of open leaves (that is, leaves with $\ell(v) \in N$) in the input derivation tree; applies $:: \mathcal{DT} \times \mathcal{C} \to$ Bool returns whether or not the input constraint applies to the input derivation tree; expand $:: \mathcal{DT} \times V \times N^* \to \mathcal{DT}$ expands the input derivation tree at the given vertex with the given child node labels; new_dt $:: N \to \mathcal{DT}$ takes a syntactic category sc as input and produces a new derivation tree with a single vertex of label sc; is_normalized $:: \mathcal{DT} \to$ Bool is shorthand for whether or not the input derivation is normalized (i.e., whether or not NormalizePr or NormalizeTa is applicable); universalize_c $:: \mathcal{DT} \times V \times \mathcal{C} \to \mathcal{C}$ rewrites the input constraint, phrasing the nonterminal expressions in terms of absolute paths from the root of the current derivation tree to vertex v; and universalize_m $:: \mathcal{DT} \to \mathfrak{M}$ translates a derivation tree to a first-order logic model by creating a variable and value for each path in the input derivation tree from root to leaf.

Lemmas. Before presenting the main results, we first present some useful helper lemmas.

First, we tie Goblin's semantics, defined in Section 4, with SMT semantics to soundly enable reasoning with SMT solvers, relating Goblin's satisfaction relation $\models_{\mathcal{G}}$ with SMT satisfaction, denoted $\models_{\mathcal{T}}$ (first-order logic satisfaction with respect to any relevant theories). This gap is bridged by the universalize_m and universalize_c functions, which translate derivation trees to SMT models and Goblin constraints to SMT constraints, respectively. Inuitively, the functions perform a *flattening* where each path in the derivation tree from root to leaf gets a corresponding SMT variable. Note that the function and predicate symbols are directly translated—the functions and predicates in Goblin's constraint language have direct SMT-LIB [5] analogues.

$$\begin{aligned}
&\text{universalize_c}(dt, v, \varphi_1 \wedge \varphi_2) = \\
&\quad \text{universalize_c}(dt, v, \varphi_1) \wedge \text{universalize_c}(dt, v, \varphi_2) \\
&\text{universalize_c}(dt, v, \neg\varphi) = \\
&\quad \neg\text{universalize_c}(dt, v, \varphi) \\
&\text{universalize_c}(dt, v, p(t_1, \ldots, t_n)) = \\
&\quad p(\text{universalize_c}(dt, v, t_1), \ldots, \text{universalize_c}(dt, v, t_n)) \\
&\text{universalize_c}(dt, v, f(t_1, \ldots, t_n)) = \\
&\quad f(\text{universalize_c}(dt, v, t_1), \ldots, \text{universalize_c}(dt, v, t_n)) \\
&\text{universalize_c}(dt, v, \langle \text{nt}_1 \rangle.\langle \text{nt}_2 \rangle.\ldots.\langle \text{nt}_n \rangle) = \\
&\quad \langle \ell(\text{root}(dt)) \rangle_\ldots_\langle \ell(\text{v}) \rangle_\langle \text{nt}_1 \rangle_\langle \text{nt}_2 \rangle_\ldots_\langle \text{nt}_n \rangle
\end{aligned}$$

$$\begin{aligned}
&\text{universalize_m}(dt) = \{(p, v) \mid \\
&\quad l \in \text{leaves}(dt) \wedge p = \langle \ell(\text{root(dt)}) \rangle_\ldots_\langle \ell(\text{parent}(l)) \rangle \wedge v = \ell(l)\}
\end{aligned}$$

To tie the semantics of Goblin to the semantics of SMT, then intuitively, we must demonstrate that a Goblin constraint φ associated with some production rule in the derivation tree is satisfied by the tree exactly when some corresponding SMT model satisfies the corresponding SMT version of the constraint (where the corresponding SMT model and constraint are defined using universalize_m and universalize_c).

Lemma 6.1. *Consider an arbitrary Goblin derivation tree tr, which has an arbitrary applicable semantic constraint φ at some arbitrary vertex v. We denote the subtree of tr rooted at v by tr'. Then,*

$$tr' \models_{\mathcal{G}} \varphi \text{ iff } \text{universalize\_m}(tr) \models_{\mathcal{T}} \text{universalize\_c}(tr, v, \varphi).$$

$$\textsc{Decide}\ \frac{\begin{array}{c}\mathsf{is\_normalized}(DT) \quad \mathsf{depth}(DT) \leq L \quad v \in \mathsf{open\_leaves}(DT) \quad \ell(v), NTs, D \notin \mathsf{visited}(DT, v) \\ \mathsf{num\_expansions}(DT) - |\mathsf{visited}(DT, v)| > 1 \quad \{\varphi \in C \mid \mathsf{applies}(DT, \varphi) \wedge \neg(\varphi \in A)\} = \emptyset \quad D^* = \mathsf{universalize\_c}(DT, v, D)\end{array}}{\begin{array}{c}DT' \leftarrow \mathsf{expand}(DT, v, NTs) \quad C' \leftarrow C \cup D^* \quad \mathsf{visited}' \leftarrow \mathsf{add}(\mathsf{visited}, (DT, v), (\ell(v), NTs, D)) \\ DS' \leftarrow DT :: DS \quad O' \leftarrow \mathsf{push}(O) \quad \mathsf{depth}' \leftarrow \mathsf{depth}[\mathsf{DT}' \mapsto \mathsf{depth}(DT) + 1]\end{array}}$$

$$\textsc{Propagate}\ \frac{\begin{array}{c}\mathsf{is\_normalized}(DT) \quad \mathsf{depth}(DT) \leq L \quad v \in \mathsf{open\_leaves}(DT) \quad \ell(v), NTs, D \notin \mathsf{visited}(DT, v) \\ \mathsf{num\_expansions}(DT) - |\mathsf{visited}(DT, v)| = 1 \quad \{\varphi \in C \mid \mathsf{applies}(DT, \varphi) \wedge \neg(\varphi \in A)\} = \emptyset \quad D^* = \mathsf{universalize\_c}(DT, v, D)\end{array}}{\begin{array}{c}DT' \leftarrow \mathsf{expand}(DT, v, NTs) \quad C' \leftarrow C \cup D^* \quad \mathsf{visited}' \leftarrow \mathsf{add}(\mathsf{visited}, (DT, v), (\ell(v), NTs, D)) \\ \mathsf{depth}' \leftarrow \mathsf{depth}[\mathsf{DT}' \mapsto \mathsf{depth}(DT) + 1]\end{array}}$$

$$\textsc{NormalizePR}\ \frac{v \in \mathsf{open\_leaves}(DT) \quad \mathsf{depth}(DT) \leq L \quad |\{\ell(v), NTs, D \in R\}| = 1 \quad D^* = \mathsf{universalize\_c}(DT, v, D)}{DT' \leftarrow \mathsf{expand}(DT, v, NTs) \quad C' \leftarrow C \cup D^*}$$

$$\textsc{NormalizeTA}\ \frac{v \in \mathsf{open\_leaves}(DT) \quad \mathsf{depth}(DT) \leq L \quad \ell(v), \tau \in \Gamma}{DT' \leftarrow \mathsf{expand}(DT, v, [])} \qquad \textsc{Assert}\ \frac{\varphi \in C \quad \varphi \notin A \quad \mathsf{applies}(DT, \varphi)}{O' \leftarrow \mathsf{assert}(O, \{\varphi\}) \quad A' \leftarrow A \cup \{\varphi\}}$$

$$\textsc{BacktrackUnsat}\ \frac{\mathsf{check\_sat}(O) = \textsc{Unsat} \quad DS = dt :: DS_2 \quad V_b = \mathsf{vertices}(DT) - \mathsf{vertices}(dt)}{\begin{array}{c}DT' \leftarrow dt \quad DS' \leftarrow DS_2 \quad O' \leftarrow \mathsf{pop}(O, 1) \quad A' \leftarrow \emptyset \\ C' \leftarrow C - \{\mathsf{universalize\_c}(DT, v', \varphi) \mid \ell(v'), \_, \varphi \in R \wedge v' \in V_b\}\end{array}}$$

$$\textsc{BacktrackDepth}\ \frac{\mathsf{depth}(DT) > L \quad DS = dt :: DS_2 \quad V_b = \mathsf{vertices}(DT) - \mathsf{vertices}(dt)}{\begin{array}{c}DT' \leftarrow dt \quad DS' \leftarrow DS_2 \quad O' \leftarrow \mathsf{pop}(O, 1) \quad bd?' \leftarrow \mathtt{true} \quad A' \leftarrow \emptyset \\ C' \leftarrow C - \{\mathsf{universalize\_c}(DT, v', \varphi) \mid \ell(v'), \_, \varphi \in R \wedge v' \in V_b\}\end{array}}$$

$$\textsc{Solve}\ \frac{\mathsf{open\_leaves}(DT) = \emptyset \quad \{\varphi \in C \mid \mathsf{applies}(DT, \varphi) \wedge \neg(\varphi \in A)\} = \emptyset \quad \mathsf{get\_model}(O) = \mathcal{M} \quad \mathsf{depth}(DT) \leq L}{\mathcal{DT}' \leftarrow \{\mathsf{instantiate}(DT, \mathcal{M})\}}$$

$$\textsc{RestartUnsat}\ \frac{\mathsf{check\_sat}(O) = \textsc{Unsat} \quad DS = [] \quad bd?}{L' \leftarrow L + 1 \quad DT' \leftarrow \mathsf{new\_dt(S)} \quad O' \leftarrow \mathsf{push}(\mathsf{reset}(O)) \quad C', A' \leftarrow \emptyset \quad bd?' \leftarrow \mathtt{false} \quad \mathsf{visited}' \leftarrow \{\}}$$

$$\textsc{RestartDepth}\ \frac{v \in \mathsf{open\_leaves}(DT) \quad \mathsf{depth}(DT) > L \quad DS = []}{L' \leftarrow L + 1 \quad DT' \leftarrow \mathsf{new\_dt(S)} \quad O' \leftarrow \mathsf{push}(\mathsf{reset}(O)) \quad C', A' \leftarrow \emptyset \quad bd?' \leftarrow \mathtt{false} \quad \mathsf{visited}' \leftarrow \{\}}$$

$$\textsc{Fail}\ \frac{\mathsf{check\_sat}(O) = \textsc{Unsat} \quad DS = [] \quad \neg bd?}{\bot}$$

Proof. Assume φ includes some arbitrary dot notation term t. We denote the Goblin interpretation of t in tr' by $\mathcal{I}_{tr'}(t)$ (as in Section 4), and analogously, we denote the SMT interpretation of the universalized version of t by $\mathcal{I}_{\mathsf{universalize\_m}(tr)}(\mathsf{universalize\_c}(tr, v, t))$. We argue that

$$\mathcal{I}_{tr'}(t) = \mathcal{I}_{\mathsf{universalize\_m}(tr)}(\mathsf{universalize\_c}(tr, v, t)).$$

The left-hand side is defined in the Goblin semantics as the value in tr' at the leaf node l found along the path (from root note v) denoted by t. The right-hand side is defined as the value of the variable whose name was constructed by serializing the path from the root of dt to v, and from v to the corresponding leaf node denoted by t. Notice that this leaf node is precisely the same as l, hence, the values are equal.

We argued that the interpretations assigned to dot notation terms is equivalent between Goblin semantics and SMT semantics (using our universalization functions). Besides dot notation, *for applicable constraints*, the satisfaction relations $\models_{\mathcal{G}}$ is defined equivalently to $\models_{\mathcal{T}}$. Hence, the claim holds. □

Using this lemma, we can identify derivation tree constraints taken from R with universalized SMT constraints produced by the universalize_c assumption (see Decide, Propagate, NormalizePr, and NormalizeTa, which translate the derivation tree constraints to SMT constraints with universalize_c before adding them to the constraint set). Further, we can identify SMT models with derivation tree leaf values (see Solve, which instantiates the SMT model from O into DT). Note that with the universalization functions, we avoid name clashes between variables in semantic constraints associated with nonterminals that are explored multiple times in the same derivation.

Next, we present two lemmas to couple the state of O to DT in every configuration (intuitively, the assertions active in O are exactly those that are relevant to DT).

Lemma 6.2. *For every configuration of every derivation, every active assertion in O is applicable in DT.*

Proof. The relevant rules are those that introduce new assertions or shrink DT, namely, Assert, BacktrackUnsat, BacktrackDepth, RestartUnsat, and RestartDepth. We proceed by structural induction on the rules, mentioning only these relevant cases. (1) Assert only adds assertions that are applicable in DT from the third precondition. (2) BacktrackUnsat and BacktrackDepth both update DT to be the old head of DS and pop O. Now notice that the only rule which pushes to the decision stack or pushes an assertion level is Decide. By the inductive hypothesis, when DT was pushed to the decision stack by Decide, O's active assertions were all applicable to DT. Notice that every push to DS has a corresponding push to O (see Decide), and analogously for pops (see BacktrackDepth and BacktrackUnsat). Hence, in the backtrack rules, when we pop to retrieve DT from DS while simultaneously popping O, we return exactly to the same state of O from when DT was pushed. Therefore, all the asserted constraints are applicable. (3) RestartUnsat and RestartDepth both trivially satisfy the claim, as they reset the assertion stack (popping all assertions). □

Lemma 6.3. *For every configuration of every derivation, every constraint from the input grammar which is applicable in DT is in C.*

Proof. New expansions of DT involving new production rule constraints only occur in Decide, Propagate, and NormalizePr. In all cases, all constraints associated with the chosen production rule (including the applicable ones) are added to C. □

Lemma 6.4. *For input grammar G, for every configuration of every derivation, $DT \in \mathcal{L}_{DT}(G)$.*

Proof. By structural induction on the rules. The relevant cases are Decide, Propagate, NormalizePr, and NormalizeTa. In Decide, Propagate, and NormalizePr, if $DT \in \mathcal{L}_{DT}(G)$, then $DT' \in \mathcal{L}_{DT}(G)$ since DT' is constructed by expanding DT according to one of the production rules in R. In NormalizeTa, if $DT \in \mathcal{L}_{DT}(G)$, then $DT' \in \mathcal{L}_{DT}(G)$ since DT is expanded by filling in a symbolic leaf value (i.e., an uninstantiated leaf with $\ell(v) = \text{None}$) for some vertex associated with a symbolic terminal in Γ. The other cases do not expand DT, so we trivially get that $DT' \in \mathcal{L}_{DT}(G)$. □

Finally, we present lemmas that are directly relevant for refutation and solution soundness (respectively).

Lemma 6.5. *Every rule except for Solve and BacktrackDepth preserves models.*

Proof. By cases. (1) Decide updates DT by making a new decision and expanding an open leaf according to a production rule option in R. The models associated with DT can be partitioned into those that use this expansion for the open leaf, and those that use a different expansion. The prior set is still captured in DT', while the latter set are captured by the new head of DS', namely, DT. (2) Propagate, NormalizePr, and NormalizeTa both expand DT without altering the decision stack. Since the expansions are forced (a precondition is that there are no alternative expansions), they cannot lose models. (3) Assert does not directly alter DT or DS, so it only relevant through Lemmas 6.3 and 6.2, which are used in the Solve case. (4) BacktrackUnsat discards DT, transfers the head of DS to DT', and maintains the rest of DS. We know from Lemma 6.2 that all the current constraints are relevant for DT, and yet the constraint set is unsatisfiable. Hence, DT cannot be extended to any models, so it is safe to discard. Since the rest of the derivation trees in DS are shifted/maintained, no models are lost. (5) RestartUnsat removes everything but the root from DT, and it does not change DS, so it cannot possibly remove models. (6) RestartDepth proceeds by the same argument as RestartUnsat (7) For Fail, We know from Lemma 6.2 that all the current constraints are relevant for DT, and yet the constraint set is unsatisfiable. Hence, DT cannot be extended to any models. But further, DS is empty, so there cannot be any models in the unprimed configuration. Since there are no models in the unprimed configuration, losing models is impossible. □

Below, we use *productive* to denote that a nonterminal is capable of deriving some finite string.

Lemma 6.6. *For an input ACFG G such that every nonterminal is productive, if the calculus does not terminate at depth limit n, it enumerates all trees of depth $\leq n$ which could possibly extend to a model before moving on to depth $n + 1$.*

Proof. The calculus implements an iterative deepening search, which exhaustively explores the search space of derivation trees $\leq L$ using Decide to store decisions to be backtracked later, BacktrackDepth to enforce the depth limit, and RestartDepth to increase the depth limit once the current limit has been thoroughly explored.

First, we argue by cases that IDS always *makes progress* in the sense that it is impossible to have an infinite chain of rule applications that does not include Decide. Assert relies on applications of Decide to collect formulas in C to assert. BacktrackDepth and BacktrackUnsat rely on decisions to backtrack. Solve always produces a saturated conclusion (argued in Theorem 6.7). RestartUnsat relies on BacktrackDepth for the precondition involving *bd?*, which in turn relies again on Decide. Each Propagate relies on at least one application of Decide (since the last Propagate) because the preconditions of Propagate entail that all but one expansion for a given DT has been explored (through Decide). RestartDepth, NormalizePr, and/or NormalizeTa could, in principle, chain indefinitely without allowing an application of Decide. However, this is impossible with our assumption that each nonterminal is productive.

Second, notice that (from Theorems 6.8 and 6.7) that saturated conclusions must come from applications of Solve and refutations must come from applications of Fail (in other words, the IDS cannot "get stuck" during enumeration).

Notice that calculus does not enumerate every possible derivation tree due to the presence of BacktrackUnsat and RestartUnsat, which discard derivation trees without exploring every possible expansion. However, whenever these rules are invoked, the constraint set is in an unsatisfiable state. Since the constraint set is applicable to DT by Lemma 6.2, no extension of DT could possibly lead to a model. Thus, it is safe to discard DT and refrain from exploring its possible extensions.

Since IDS always makes progress and all other derivation trees are enumerated by IDS, the claim holds. □

Theorems. We now present the main theorems.

Theorem 6.7. *(Refutation soundness) Every refutation has an initial configuration associated with an unsatisfiable grammar.*

Proof. First, we argue that no refutation can include an application of Solve. Assume for a contradiction that Solve is applied somewhere in a refutation. Then, the conclusion of this Solve rule must not be a saturated configuration, because the proof tree must have $\bot$ at every leaf. However, this is impossible, as after applying Solve, none of the other rules can be applicable: Decide, Propagate, NormalizePr, and NormalizeTa require some open leaf, but one cannot exist (see the preconditions to Solve). Assert requires the existence of some $\varphi \in C$ such that φ is applicable but not in A, but this is also impossible due to a precondition of Solve. BacktrackUnsat requires that the current asserted constraint set by unsatisfiable, but this cannot be the case since O just returned a model from check_sat. BacktrackDepth requires the depth of DT to be above L, but this contradicts a precondition of Solve. RestartUnsat and RestartDepth proceed with the same arguments as BacktrackUnsat and BacktrackDepth. Fail proceeds with the same argument as BacktrackUnsat.

Second, we argue that every refutation containing BacktrackDepth can be rewritten to a different refutation that does not contain BacktrackDepth. (We do this because BacktrackDepth does not necessarily preserve models.) Notice that due to the precondition on *bd?*, no proof tree can include an application of BacktrackDepth, followed by an application of Fail, without an application of RestartDepth or RestartUnsat in between. Therefore, consider an arbitrary proof tree containing BacktrackDepth. Then, consider the last application of RestartDepth or RestartUnsat, and take the subtree rooted at the conclusion of this application as the new refutation. (This conclusion is a valid initial configuration, just with a higher depth limit.)

Since Solve and BacktrackDepth are never present, we can argue from structural induction on proof trees using Lemma 6.5 that any proof tree with a leaf containing $\bot$ must also have an unsatisfiable root. □

Theorem 6.8. *(Solution soundness) Every solution has an initial configuration associated with a satisfiable grammar.*

Proof. We argue that every saturated leaf of a solution corresponds to a model of the input grammar. The arguments proceeds in three steps. We argue that (1) the conclusion of an application to Solve must be saturated, (2) the conclusion of an application of Solve corresponds to a model, and (3) the conclusion of any rule except for Solve cannot be saturated.

We already argued step (1) in Theorem 6.7.

For step (2), we can directly take DT' as the model. We know DT' is syntactically valid from Lemma 6.4, and we know it is semantically valid with respect to all the applicable constraints from Lemmas 6.3, 6.2, and 6.1, along with the precondition that all applicable constraints in C must be asserted since the last pop (second precondition). Finally, the inapplicable constraints are trivially satisfied, according to the semantics in Section 4.

We argue step (3) by contradiction. Assume the conclusion of another rule is saturated. First, the depth of DT must be $\leq L$, or else BacktrackDepth or RestartDepth would be applicable. Second, the derivation tree cannot have any open leaves, or else Decide, Propagate, NormalizePr, or NormalizeTa would be applicable (assuming DT has depth $\leq L$). Also, the derivation tree cannot have any applicable but unasserted constraints, or else Assert would be applicable. Finally, the current constraint set cannot be unsatisfiable, or else BacktrackUnsat or RestartUnsat would be applicable. However, since all of the above hold, it must be the case that Solve is applicable, a contradiction with the assumption that the conclusion was saturated. □

In solution completeness, note that the precondition of nonterminal productivity is enforced in Goblin by a syntactic check before the main solve algorithm.

Theorem 6.9. *(Solution completeness) For every satisfiable grammar G such that every nonterminal is productive, there exists a solution with an initial configuration built from G.*

Proof. The result directly follows from Lemma 6.6. □

Notably, the calculus is refutationally incomplete in the sense that there might not exist a refutation for unsatisfiable initial configurations. This is unsurprising, as the unsatisfiability of the initial configuration may be an inductive property (e.g., a grammar denoting a list of integers whose individual elements are negative but whose total sum must be positive). Refutations are not discussed in prior work [31], so the presence of refutations and refutation soundness in our calculus is an improvement over prior work, despite the lack of refutation completeness.

7 CLP Semantics

We present an alternative language semantics for Goblin as a *translational semantics* to CLP problems. While the semantics described in Section 4 works naturally with our proofs, the CLP semantics is also useful to get a sense for how Goblin relates to CLP. More concretely, we define a translation of an input ACFG to a CLP problem, which we model as a set of constrained horn clauses (CHCs). Each CHC is of the form $H \leftarrow B_1, \ldots, B_n, \phi$, where H and B_1 through B_n are terms, and ϕ is a constraint over the free variables from the prior terms. Assuming the existence of a CLP solver that can handle all the constraints in the generated CLP problem, the CLP semantics can be viewed as a path toward an alternative, CLP-based engine for Goblin.

We have

$$[\![G]\!] = \bigcup_{nt,N,C \in R} [\![nt, N, C]\!]_{\text{pr}} \cup \bigcup_{nt,\tau \in \Gamma} [\![nt, \tau]\!]_{\text{ta}},$$

where

$$[\![nt, \tau]\!]_{\mathsf{ta}} = \{nt(NT) \leftarrow NT \in \tau\}$$

$$[\![nt, N, C]\!]_{\mathsf{pr}} = \{[\![nt, N, C]\!]_{\mathsf{lhs}} \leftarrow [\![nt, N, C]\!]_{\mathsf{rhs}}\} \cup \bigcup_{e \in \mathsf{nt\_exprs}(C)} [\![e]\!]_{\mathsf{fe}}$$

$$[\![nt, N, C]\!]_{\mathsf{lhs}} = nt(nt_i(N_1, \ldots, N_m)), \text{ where each } N_j \in N \text{ for the } i\text{th production rule option for } nt$$

$$[\![nt, N, C]\!]_{\mathsf{rhs}} = \bigwedge_{m \in N} m(M_i) \wedge \bigwedge_{e \in \mathsf{nt\_exprs}(C)} [\![e]\!]_{\mathsf{nt\_expr}} \wedge [\![C]\!]_{\mathsf{b\_expr}} \text{ for the } i\text{th occurrence of } m \text{ in } N$$

$$[\![nt_1.nt_2 \ldots nt_n]\!]_{\mathsf{fe}} = \{[\![nt_1.nt_2 \ldots nt_n]\!]_{\mathsf{fe\_lhs}} \leftarrow [\![nt_1.nt_2 \ldots nt_n]\!]_{\mathsf{fe\_rhs}}\} \text{ for each of } nt_1\text{'s production rules} \cup [\![nt_2 \ldots nt_n]\!]_{\mathsf{fe}}$$

$$[\![nt_1.nt_2 \ldots nt_n]\!]_{\mathsf{fe\_lhs}} = nt_1(nt_{1,i}(N_1, \ldots, N_m), nt_n), \text{ where each } N_j \in N \text{ for the } i\text{th production rule option for } nt_1$$

$$[\![nt_1.nt_2 \ldots nt_n]\!]_{\mathsf{fe\_rhs}} = \bigwedge [\![nt_2 \ldots nt_n]\!]_{\mathsf{nt\_expr}} \wedge \mathsf{append}(\mathsf{nt}_{\mathsf{n\_1}}, \ldots, \mathsf{nt}_{\mathsf{n\_i}})$$

$$[\![nt_1.nt_2]\!]_{\mathsf{fe}} = \mathsf{nt}_1\_\mathsf{nt}_2(\mathsf{nt}_{1,i}(q_1, \ldots, q_n), [NT_{2,1}, \ldots, NT_{2,m}]) \text{ for the } i\text{th production rule option for } \mathsf{nt}_1, \text{ where each } q_j \text{ is } \mathsf{NT}_{2,k} \text{ if the } j\text{th nonterminal in the } i\text{th production rule for } \mathsf{nt}_1 \text{ is } \mathsf{nt}_2, \text{ and \_ otherwise, with } k \text{ ranging from 1 to } m$$

$$[\![nt_1 \ldots nt_n]\!]_{\mathsf{nt\_expr}} = nt_{1\_} \ldots \_nt_n(NT_{1,i}, NT_{1\_} \ldots \_NT_{n,i})$$

$$[\![C]\!]_{\mathsf{b\_expr}} = \bigwedge_{c_i \in \mathsf{generalize}(C)} c_i$$

Above, generalize performs the disambiguation of dot notation constraints analogous to resolve as discussed in Section 3, and nt_exprs(.) retrieves a set of all dot notation expressions present in the input constraints. Also, we use square brackets to denote list literals and append to denote a multi-arity list append function. The pattern "_" denotes a universal match for an unused value. In the encoding, we use both lowercase and uppercase names, which tie to corresponding nonterminals in the input grammar. By convention, we use the lowercase names to denote constructor symbols and the uppercase names to denote variables (e.g. nt and NT, both referring to the same nonterminal in the input grammar).

Intuitively, we partition the CHCs into those we need for type annotations and those we need for production rules. The type annotation CHCs simply state that the given nonterminal must be a member of its given type; the production rule CHCs are more complicated. First, the production rule CHC definitions are split into the left-hand side (LHS) and right-hand side (RHS). They use CLP terms to capture the structure of the grammar and CLP constraints to capture the semantic constraints from C. In addition to these LHS and RHS definitions, each production rule also generates CHCs which we call *field extractors*, defined by $[\![.]\!]_{\mathsf{fe}}$, which enable passing variables between clauses to support the dot notation constraints. The field extractor CHCs are invoked by calls defined by $[\![.]\!]_{\mathsf{nt\_expr}}$.

8 Implementation and Evaluation

Goblin Implementation. We implemented Goblin in ~9.9K lines of OCaml 5.1.1 code using cvc5 [3] version 1.3.0 as our underlying constraint solver. The main pipeline is depicted in Figure 2.

Code availability. Goblin's source code and experimental evaluation details are available at [22].

Empirical Evaluation. We evaluate Goblin on four case studies. The first three case studies, XML, CSV, and Scriptsize C input generation, serve as a direct comparison to prior work [31] and demonstrate Goblin's capability of handling complex constraints. In these case studies, we evaluate the correctness, efficiency, diversity, and length of generated inputs. The fourth case study involves generation of valid WiFi SAE commit and confirm frames, which cannot be handled directly by prior work due to the presence of rich bit vector constraints. Rather than taking a fixed grammar and constraints (as performed by prior work and in the prior two case studies), we demonstrate how Goblin's local production rule constraints can be easily integrated into a fuzzing workflow involving *grammar-level production rule mutation operations*.

8.1 XML, CSV, and Scriptsize C

We present our main results for the XML, CSV, and Scriptsize C case studies in Table 1. We collected the results with 1-hour runs of Goblin and ISLa locally on a Macbook Pro with an M2 chip and 16GB RAM. Our main research questions are:

RQ1. **How *efficient* is Goblin compared to ISLa?**
RQ2. **How *diverse* are Goblin's outputs compared to ISLa's?**
RQ3. **How does the *size* of Goblin's outputs compare to ISLa's?**

Both case studies involve *structural constraints*. In XML, there is a *definition before use* constraint that mandates that XML namespaces must be used only in scopes in which they are defined, a *no redefinition* constraint that mandates that XML namespaces are not redefined, and a *matching tags* constraint that matching closing and opening tags have the same identifiers. Scriptsize C has analogous definition before use and no redefinition constraints. In CSV, there is a *column count* constraint that mandates that all rows in the CSV file contain the same number of columns. In ISLa, structural constraints are supported through built-in *structural predicates*. However, in Goblin's DSL, structural predicates are not built-in. *Nonetheless, these constraints can still be encoded in Goblin.* (see Section 8.2.)

Correctness. Based on our formal soundness guarantees, we can confidently claim that all generated inputs are correct with respect to the input specification. Prior work [31, 36] use *precision* as a metric for evaluation, defines as the percentage of inputs successfully parsed by the system under test. We argue this metric is not relevant, since it is a metric of the system under test rather than of the input generator. However, we note that for CSV and XML, we formulated constraints equivalent to those in prior work.

Efficiency. To address RQ1, we compare the efficiency of Goblin to ISLa in terms of the rate of input generation. Goblin outperforms ISLa by a wide margin for XML inputs and by a narrow margin for Scriptsize C inputs. We hypothesize that incremental solving boosts performance by allowing quicker backtracking and enabling the mending of partial solutions. Notably, Goblin struggles on CSV inputs. The semantic constraint, which mandates that all rows have

the same number of columns, restricts the *derivation strategy* rather than the values of the leaf terms. A current weakness of Goblin is that it does not use any special heuristic for choosing production rule expansions (only coin flips), which we believe explains the poor performance compared to ISLa.

Diversity. To address RQ2, we compare efficiency based on *accumulated k-path coverage* [16], which measures how many paths of length k from the grammar are actually present in the generated inputs, where a *path* refers to a sequence of nonterminal or terminal symbols. A higher k-path coverage is more diverse and thus better. As in [31], we present results with $k = 3$. Goblin performs very well, reaching near full coverage on every case study. However, because Goblin generates ADT terms rather than strings, we do not use any concrete terminal symbols in our encoding (terminal symbols are generated from symbolic terminals, optionally with semantic constraints), which differs from ISLa's encoding which includes concrete terminals. So, although Goblin performs well, we cannot claim that the comparison with ISLa is direct.

Length. To address RQ2, we compare the mean length of inputs generated by Goblin and ISLa. Goblin's inputs, on average, are more concise than ISLa's. We report this metric to be informative, but we argue that there is no clear benefit or drawback to having shorter or longer inputs. Further, one can always impose additional semantic constraints or grammar-level mutations to either increase or decrease the length of generated inputs.

8.2 Structural Constraint Encoding

It may be unclear how one would express structural constraints in Goblin. Intuitively, we use local production rule constraints, along with dot notation and support for rich data types (e.g., sets), to specify the desired properties akin to how one would do so with an attribute grammar or a CLP system. The encodings are available along with Goblin's source code at [22]. As a simpler example, consider a grammar encoding two lists with the structural constraint that every element in the second list must also be present in the first:

Listing 8: Structural constraint

```
<S> ::= <L1> <L2>
      { <L2>.<_defs_down> = <L1>.<_defs_up>; };
<L1> ::= <_defs_up> <str> <L1>
{ <_defs_up> = set.union(set.singleton(<str>),
                         <L1>.<_defs_up>); }
       | <_defs_up> <str>
       { <_defs_up> = set.singleton(<str>); };
<L2> ::= <_defs_down> <str> <L2>
       { set.member(<str>, <_defs_down>);
         <L2>.<_defs_down> = <_defs_down>; }
       | <_defs_down> <str>
       { set.member(<str>, <_defs_down>); };
<str> :: String;
<_defs_down> :: Set(String);
<_defs_up> :: Set(String);
```

We specify the constraint by introducing extra symbolic terminals into the grammar, preceded by underscores, to serve as means of passing context in the style of attribute grammars. We remind the reader that Goblin is designed as a minimal intermediate language, and a surface-level language with proper support for attributes and other features is left for future work.

Tool	Subject	Constraints	Efficiency (inputs/min)	k-Path coverage	Mean length
Goblin	Scriptsize C	def-use; redef;	108.53	95.4	18.75
ISLa	Scriptsize C	def-use; redef;	101.70	50.72	24.01
Goblin	XML	def-use; redef; tags	100.00	84.44	11.71
ISLa	XML	def-use; redef; tags	48.00	75.37	38.03
Goblin	CSV	col count	19.5	100.00	88.45
ISLa	CSV	col count	91.83	100.00	487.08

Table 1: Results for each subject, tool, and associated metrics.

8.3 WiFi: WPA3-SAE Handshake Frames

We now present the results of our WiFi authentication case study, focusing on the state-of-the-art WPA3-SAE handshake [19]. The handshake has complex packet formats consisting of (1) rich bit vector constraints, (2) inter-field dependencies, and (3) non-linear cryptographic elements. We instantiate Goblin with an existing grammar-based black-box fuzzer Saecred [11] that generates context-sensitive SAE commit and confirm frames. Saecred reduces input generation to a SyGuS problem and uses a SyGuS solver [1] to generate byte sequences satisfying the grammar constraints. To evaluate Goblin, we replace Saecred's input generator with Goblin (referred to as Goblin-Saecred) and compare it with the original Saecred fuzzer.

Evaluation setup. We run Goblin-Saecred and Saecred for 24 hours each on an Intel Core Ultra 9 185H with 64 GB RAM. The black-box target was `hostapd` v2.10, a popular open-source WiFi access point software. We use the same grammar and constraints for both fuzzers, which we obtained from the Saecred authors.

Evaluation criteria. We propose three evaluation criteria: (1) *performance* – we measure the time taken for Goblin and SyGuS to generate byte sequences from the mutated grammar, (2) *expressiveness* – we measure the number of grammar-level mutations Goblin and SyGuS can handle, and (3) *effectiveness* – we measure the number of bugs uncovered in `hostapd` by the two fuzzers.

Performance. Both variations of Saecred were able to generate 97, 182 mutated grammars for byte serialization. Out of those, 49, 460 passed the Goblin language-emptiness checks. Figure 4 shows the (log-scaled) smoothed time taken by Goblin-Saecred and SyGuS-Saecred to generate byte sequences from the 49, 460 mutated grammar instances. Goblin-Saecred outperforms SyGuS-Saecred by a wide margin. Table 2 shows the summary statistics of the time taken to generate the byte sequences. Goblin-Saecred is 36x faster on average than SyGuS-Saecred.

Expressiveness. Out of the 97, 182 mutated grammar instances, some instances were unsolvable due to the presence of unsatisfiable constraints; however, the expressive capabilities of both approaches were vastly different. Table 2 shows that Goblin-Saecred was able to handle 49, 460 grammar instances whereas SyGuS-Saecred was

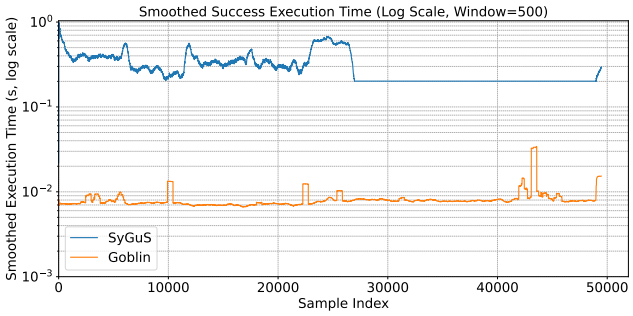

Figure 4: Performance comparison of Goblin-Saecred and SyGus-Saecred on WiFi SAE commit and confirm frames. The x-axis shows the grammar instance, and the y-axis shows the log-scale time taken to produce byte serializations.

Metric / Method	Goblin	SyGuS	Unit
Successes	49,460	23,904	count
Failures	47,722	73,278	count
Total Grammars: 97,182			
Mean Time	0.0082	0.2968	seconds
Median Time	0.0074	0.2014	seconds
Std. Dev. Time	0.0588	0.3606	seconds

Table 2: Comparison of Goblin-Saecred and SyGuS-Saecred in terms of success/failure count and execution time. Goblin on average performs 36x faster than SyGuS.

only able to handle 23, 904 grammar instances, leading to 73, 278 failures as opposed to Goblin-Saecred's 47, 722. This shows that Goblin is able to handle a wider variety of grammar-level mutations than SyGuS-based input generator of Saecred.

Effectiveness. The 24 hour fuzzing campaigns resulted in Goblin-Saecred uncovering all 4 known bugs from the original Saecred paper [11], including denial-of-service and downgrade attacks.

9 Related Work

The most relevant related work was discussed in Sections 1 and 2: context-free grammar-based fuzzing [17], ISLa and Fandango [31, 36], and system-specific input generators [7, 34].

CLP. Our approach is reminiscent of constraint logic programming (CLP), which extends logic programming to also handle constraint satisfaction. In fact, Dewey et al. [13] presented an approach for CLP-based language fuzzing and applied it to JavaScript programs. In principle, Goblin can be framed in terms of CLP—in Section 7, we define an alternative semantics of Goblin in terms of a CLP problem. However, we cannot rely on CLP solvers in practice, as they do not natively support arbitrary SMT theories (e.g., bit vectors, strings) or their combinations.

SyGuS. Another related formalism is syntax-guided synthesis (SyGuS) [1], which presents a synthesis problem in terms of a function to synthesize that must be in the language of some user-defined grammar while also satisfying user-defined semantic constraints. In principle, one could use a SyGuS engine (e.g., cvc5SY [30]) as the backend reasoner for a tool like Goblin (as in Saecred [11]). But, there are notable obstacles. For recursive grammars, in order to specify that a constraint applies to *all* instances of a production rule expansion during a derivation, the corresponding SyGuS constraint must be expressed as a recursive function, which is nontrivial to synthesize and often suffers from performance problems in a SyGuS setting. Further, by default, SyGuS solvers only support semantic constraints from the top level (i.e., predicates of type $\tau \rightarrow$ Bool, where τ is the type of the function to synthesize). Hence, supporting local production rule semantic constraints in general is nontrivial.

Attribute grammars. Attribute grammars [21] are a formalism built on top of context-free grammars, where the grammar can additionally assign *attributes* to each production rule. Attributes are additional variables that can be used to communicate context-sensitive information throughout a parse tree, and can be defined in terms of descendants (called *synthesized attributes*) or ancestors (called *inherited attributes*). One can use attributes to express and check semantic properties; however, prior work (e.g., [32]) on attribute grammars has focused on *parsing* rather than *generation*.

Property-based testing. Property-based testing (PBT), pioneered by QuickCheck [8], is a testing strategy in which the programmer annotates expected invariants in the source code, and a testing tool generates inputs to check whether or not the invariants are ever falsified. These approaches differ from the one employed by Goblin in two main ways: (*i*) Goblin operates with system-level input rather than internal inputs, and (*ii*) Goblin allows for the annotation of rich semantic constraints and employs heuristics that are more in depth than *generate-and-test* approaches typical in PBT.

Context-sensitive parsing. In general, most work on grammars with context-sensitive constraints has focused on parsing rather than input generation. For example, Parsley [24] is a tool that uses an input language similar in spirit to Goblin's, but it is a tool for synthesizing verified parsers for languages with context-sensitive constraints (rather than for input generation).

10 Conclusion

We presented Goblin, a system for generating context-sensitive inputs. Compared to prior work, Goblin is novel in its capability to handle constraints from arbitrary SMT theories, its generality despite using a minimal core language, and its amenability to integration with grammar-based fuzzing workflows due to its semantics of local production rule constraints.

Limitations. Compared to prior works, ours is most similar in spirit to ISLa [31], and we experience a similar set of tradeoffs. For non-derived fields, the constraint language space is restricted to being easily translatable to SMT-LIB [5]. Moreover, constraint solving can be expensive for some inputs.

User-facing language. Goblin does not yet have a high-level, user-facing language. We expect such a language to be designed such that (*i*) one can specify structural constraints (as in Listing 8) without ever needing to introduce extra syntactic categories into the grammar, and (*ii*) one can use built-in functions to express common constraint patterns (e.g., a `length(.)` function that computes the length of its input, where the input can be any syntactic category). In particular, these built-ins should be designed such that they are

legal in either derived fields *or* general semantic constraints (the formal is trivial but restrictive, while the latter is more general but requires more insight).

Relationship with LLMs. One could query a language model for the problem addressed by Goblin (i.e., input generation with a grammar input). One on hand, Goblin is locally executable, which comes with latency and cost advantages. Additionally, Goblin's search algorithm is provably sound, while LLMs may hallucinate. However, for certain expensive problems where Goblin's constraint solving is nonterminating, LLMs may still be able to generate valid instances. One could envision approaches combining the advantages Goblin and LLMs (e.g., generating untrustworthy instances with an LLM and repairing them with Goblin).

References

[1] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*. IEEE, Institute of Electrical and Electronics Engineers (IEEE), 1–8.

[2] John W Backus. 1959. The syntax and the semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference. In *ICIP Proceedings*. Butterworths, London, 125–132.

[3] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, et al. 2022. cvc5: A versatile and industrial-strength SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, Springer, 415–442.

[4] Clark Barrett, Roberto Sebastiani, Sanjit Seshia, and Cesare Tinelli. 2009. Satisfiability Modulo Theories. In *Handbook of Satisfiability*, Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh (Eds.). Vol. 185. IOS Press, Chapter 26, 825–885.

[5] Clark Barrett, Aaron Stump, and Cesare Tinelli. 2010. The SMT-LIB Standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, A. Gupta and D. Kroening (Eds.).

[6] Clark Barrett and Cesare Tinelli. 2018. Satisfiability modulo theories. In *Handbook of model checking*, Edmund Clarke, Thomas Henzinger, Helmut Veith, and Roderick Bloem (Eds.). Springer, 305–343.

[7] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. 2014. Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In *2014 IEEE Symposium on Security and Privacy*. IEEE, IEEE Computer Society, 114–129.

[8] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. Association for Computing Machinery, 268–279.

[9] Bruno Courcelle and Paul Franchi-Zannettacci. 1982. Attribute grammars and recursive program schemes I. *Theoretical Computer Science* 17, 2 (1982), 163–191.

[10] cvc5 Developers. 2025. cvc5 Nonstandard Theories. https://cvc5.github.io/docs/cvc5-1.0.2/theories/theories.html#non-standard-or-extended-theories.

[11] Muhammad Daniyal Pirwani Dar, Rob Lorch, Aliakbar Sadeghi, Vincenzo Sorcigli, Héloïse Gollier, Cesare Tinelli, Mathy Vanhoef, and Omar Chowdhury. 2025. Saecred: A State-Aware, Over-the-Air Protocol Testing Approach for Discovering Parsing Bugs in SAE Handshake Implementations of COTS Wi-Fi Access Points. In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE, IEEE, 3691–3709.

[12] Martin Davis, George Logemann, and Donald Loveland. 1962. A Machine Program for Theorem-Proving. *Commun. ACM* 5, 7 (1962), 394–397. doi:10.1145/368273.368557

[13] Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2014. Language fuzzing using constraint logic programming. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. Association for Computing Machinery, 725–730.

[14] Matheus E Garbelini, Vaibhav Bedi, Sudipta Chattopadhyay, Sumei Sun, and Ernest Kurniawan. 2022. {BrakTooth}: Causing havoc on bluetooth link manager via directed fuzzing. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, 1025–1042.

[15] Matheus E Garbelini, Chundong Wang, and Sudipta Chattopadhyay. 2020. Greyhound: Directed greybox wi-fi fuzzing. *IEEE Transactions on Dependable and Secure Computing* 19, 2 (2020), 817–834.

[16] Nikolas Havrikov and Andreas Zeller. 2019. Systematically covering input structure. In *2019 34th IEEE/ACM international conference on automated software engineering (ASE)*. IEEE, IEEE Press, 189–199.

[17] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *21st USENIX Security Symposium (USENIX Security 12)*. USENIX Association, 445–458.

[18] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. 2001. Introduction to automata theory, languages, and computation. *Acm Sigact News* 32, 1 (2001), 60–65.

[19] IEEE Std 802.11. 2020. Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Spec.

[20] Imtiaz Karim, Fabrizio Cicala, Syed Rafiul Hussain, Omar Chowdhury, and Elisa Bertino. 2020. ATFuzzer: dynamic analysis framework of AT interface for Android smartphones. *Digital Threats: Research and Practice* 1, 4 (2020), 1–29.

[21] Donald E Knuth. 1968. Semantics of context-free languages. *Mathematical systems theory* 2, 2 (1968), 127–145.

[22] Robert Lorch, Muhammad Daniyal Pirwani Dar, Cesare Tinelli, and Omar Chowdhury. 2025. Parse this! Summoning Context-Sensitive Inputs with Goblin. https://github.com/lorchrob/goblin_full_version.

[23] Valentin JM Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2312–2331.

[24] Prashanth Mundkur, Linda Briesemeister, Natarajan Shankar, Prashant Anantharaman, Sameed Ali, Zephyr Lucas, and Sean Smith. 2020. Research Report: The Parsley Data Format Definition Language. In 2020 IEEE Security and Privacy Workshops (SPW). 300–307.

[25] NIST. 2023. CVE-2021-1903. https://nvd.nist.gov/vuln/detail/cve-2023-29548.

[26] NIST. 2023. CVE-2021-30247. https://nvd.nist.gov/vuln/detail/cve-2021-30247.

[27] NIST. 2023. CVE-2023-29548. https://nvd.nist.gov/vuln/detail/cve-2023-29548.

[28] NIST. 2024. CVE-2024-52924. https://nvd.nist.gov/vuln/detail/CVE-2024-52924.

[29] Jon Postel. 1981. *Internet Protocol*. Technical Report. Internet Engineering Task Force. https://www.rfc-editor.org/rfc/rfc791

[30] Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark Barrett, and Cesare Tinelli. 2019. cvc4sy: Smart and fast term enumeration for syntax-guided synthesis. In *International Conference on Computer Aided Verification*. Springer, Springer, 74–83.

[31] Dominic Steinhöfel and Andreas Zeller. 2022. Input invariants. In *Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering*. Association for Computing Machinery, 583–594.

[32] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. 2010. Silver: An extensible attribute grammar system. *Science of Computer Programming* 75, 1-2 (2010), 39–54.

[33] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, IEEE, 724–735.

[34] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. Association for Computing Machinery, 283–294.

[35] Michal Zalewski. 2014. *American Fuzzy Lop (AFL)*. Technical Report. https://lcamtuf.coredump.cx/afl/.

[36] José Antonio Zamudio Amaya, Marius Smytzek, and Andreas Zeller. 2025. FANDANGO: Evolving Language-Based Testing. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA040 (June 2025), 23 pages. doi:10.1145/3728915

[37] Jialun Zhang, Greg Morrisett, and Gang Tan. 2023. Interval parsing grammars for file format parsing. *Proceedings of the ACM on Programming Languages* 7, PLDI (2023), 1073–1095.