Specifying Input Constraints

Dominic Steinhöfel
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
dominic.steinhoefel@cispa.de

Andreas Zeller
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
zeller@cispa.de

ABSTRACT

Grammar-based fuzzers are highly efficient in producing syntactically valid system inputs. However, as context-free grammars cannot capture *semantic* input features, generated inputs will often be rejected as semantically invalid by a target program. We propose ISLa, a *declarative specification language for context-sensitive properties* of structured system inputs based on context-free grammars. With ISLa, it is possible to specify *input constraints* like "a location has to be initialized before its use," "the length of the 'file name' block in a tar file has to equal 100 bytes," or "the number of columns in all rows must be identical."

ISLa constraints can be used for *parsing* or *validation* ("Does an input meet the expected constraint?") as well as for *fuzzing*, since we provide both an *evaluation* and *input generation component*. ISLa embeds SMT formulas as an island language, leveraging the power of decision procedures about string theories implemented in modern solvers like Z3 to solve atomic semantic constraints. On top, it adds universal and existential *quantifiers* over the structure of derivation trees from a grammar, and structural ("X occurs before Y") and semantic ("X is the checksum of Y") *predicates*.

We formalize syntax and semantics of ISLa constraints, provide a formal description of our implemented generator and prove its soundness and completeness. In our evaluation, we demonstrate that a few ISLa constraints suffice to produce inputs that are 100% semantically valid while still maintaining input diversity.

KEYWORDS

fuzzing, specification language, grammars, context-sensitive constraints

1 INTRODUCTION

Automated software testing with random inputs (*fuzzing*) [18] is an efficient technique for finding bugs in programs. The OSS-Fuzz project, for example, discovered over 30,000 bugs in 500 open source projects [23]. Pure random generation of input strings can quickly discover errors in input processing. But if a program expects complex structured inputs (e.g., C programs, JSON expressions, or binary formats), the chances of randomly producing *valid inputs* that are accepted by the parser and reach deeper functionality are low.

⟨*xml-tree*⟩ ::= ⟨*xml-openclose-tag*⟩
| ⟨*xml-open-tag*⟩ ⟨*inner-xml-tree*⟩ ⟨*xml-close-tag*⟩
⟨*inner-xml-tree*⟩ ::= ⟨*text*⟩ | ⟨*xml-tree*⟩
| ⟨*inner-xml-tree*⟩ ⟨*inner-xml-tree*⟩
⟨*xml-open-tag*⟩ ::= '<' ⟨*id*⟩ '>'
| '<' ⟨*id*⟩ '␣' ⟨*xml-attribute*⟩ '>'
⟨*xml-close-tag*⟩ ::= '</' ⟨*id*⟩ '>'
⟨*xml-openclose-tag*⟩ ::= '<' ⟨*id*⟩ '/>'
| '<' ⟨*id*⟩ '␣' ⟨*xml-attribute*⟩ '/>'
⟨*xml-attribute*⟩ ::= ⟨*id*⟩ '="' ⟨*text*⟩ '"'
| ⟨*xml-attribute*⟩ '␣' ⟨*xml-attribute*⟩

Figure 1: A context-free grammar for XML (excerpt)

Language-based fuzzers [8, 11, 12] overcome this limitation by generating inputs from a *specification* of a program's expected input language, frequently expressed as a *Context-Free Grammar (CFG)*. This considerably increases the chance of producing an input passing the program's parsing stage and reaching its core logic. Yet, while being great for parsing, *CFGs are often too coarse for* producing *inputs*. Consider, e.g., the language of XML documents (without document type). This language is *not* context free.[1] Still, it can be approximated by a CFG. Fig. 1 shows an excerpt of a CFG for XML. When we used a coverage-based fuzzer to produce 10,000 strings from this grammar, exactly *one* produced document (`<0 L="cmV">B7</0>`) contained a matching tag pair. This result is typical for language-based fuzzers when using them with a language specification designed for *parsing* and which therefore is more permissive than a language specification for *producing* would have to be. This is unfortunate, as hundreds of language specifications for parsing exist.

To allow for precise production, we need to enrich the grammar with more information, or switch to a totally different formalism. However, existing solutions all have their drawbacks. Using *general purpose code* to produce inputs, or enriching grammars with such code is closely tied to an implementation language, and does not allow for *parsing* and *recombining* existing inputs, which is a common feature of modern fuzzers. *Unrestricted grammars* can in principle

[1] Apply the pumping lemma with $\texttt{<a}^n\texttt{b}^n\texttt{></a}^n\texttt{b}^n\texttt{>}$.

Listing 1: ISLa constraint for well-balanced XML expressions (reference grammar: Fig. 1).

```
1 const start: <start>;
2 vars {
3     tree: <xml-tree>;
4     opid, clid: <id>;
5 }
6 constraint {
7   forall tree="<{opid}[ <xml-attribute>]><inner-xml-tree></{
        ↪ clid}>" in start:
8     (= opid clid)
9 }
```

specify any computable input property, but we see them as "Turing tar-pits," in which "everything is possible, but nothing of interest is easy" [21]—just try, for instance, to express that some number is the sum of two input elements. Finally, one could also replace grammars by a *different formalism;* but this would mean to renounce a concept for specifying inputs that is familiar to any software developer.

In this paper, we bring forward a different solution by proposing a (programming and target) *language-independent*, *declarative* specification language named ISLa (Input Specification Language) for expressing *context-sensitive constraints over CFGs*. By enriching existing grammars with constraints, we leverage the simplicity of CFGs, while permitting to significantly extend their limited expressiveness. When formalizing a new target language, one starts with the definition of a CFG (which, for many languages, might be readily available somewhere in the internet). Then, one iteratively *strengthens the definition* by adding more and more ISLa constraints until the represented language is a sufficiently close approximation of the target language. ISLa constraints can be used to *produce* inputs that are syntactically and semantically valid, but also for *validation*, i.e., to *check* whether an input is syntactically and semantically valid.

ISLa embeds SMT formulas as an island language[2], leveraging the power of decision procedures about string theories implemented in modern solvers like Z3 to solve atomic semantic constraints. On top, it adds universal and existential *quantifiers* over the structure of derivation trees from a grammar, and structural ("X occurs before Y") and semantic ("X is the checksum of Y") *predicates*.

Let us illustrate the expressive power of ISLa in an example. In Fig. 1, the IDs in ⟨*xml-open-tag*⟩ XML elements would not match the IDs in corresponding ⟨*xml-close-tag*⟩ elements. The ISLa constraint in Listing 1 solves this problem by enforcing that the two IDs match. Lines 1 to 5 define variable and constant symbols used in the constraint; Lines 6 to 9 contain the constraint itself. Each variable or constant symbol has a type, which is a nonterminal in the reference grammar (here the XML grammar in Fig. 1), specified after the colon. An ISLa constraint contains exactly one constant symbol, which determines the type of the inputs described by the constraint (here ⟨*start*⟩).

The top-level formula is a universal quantifier (**`forall`**). It quantifies over all sub expressions of type ⟨*xml*⟩, since this is the type of the variable `tree`. The quantifier uses *pattern matching*. Only matches conforming to the pattern (specified within quotation marks) are considered; in the case of a successful match, not only the quantified variable `tree`, but also the variables in the pattern (surrounded by curly braces) are *bound* to the corresponding parts of the matched input segment. Match expressions may contain optional elements surrounded by square brackets to capture multiple expansion alternatives at once. The core of the **`forall`** formula is an *SMT-LIB S-expression* stating the equality of the variables `opid` and `clid`. Since ISLa extends the SMT-LIB language [25], it supports all its string constraints.

Since ISLa constraints are closed under conjunction (**`and`**) and disjunction (**`or`**), it is easy to refine (or relax) constraints. ISLa is thus well suited for *targeted* testing, or, e.g., for describing a *specific class of inputs* that trigger a bug in a debugging scenario. Thanks to its declarative nature, it can also be used for formulating human-readable *specifications* of the expected inputs of a system.

In addition to using ISLa as standalone test generation tool, one can *integrate it with existing evolutionary fuzzers* such as AFL[3]. As a start, high-quality inputs produced by ISLa can be used as *seed inputs* which most evolutionary fuzzers depend on to get going. ISLa specifications (and ISLa's constraint checker) can be integrated into the *fitness function* of evolutionary fuzzers, guiding their mutations towards semantically valid inputs. Finally, ISLa can be used as a fast *checker* of inputs, rejecting invalid inputs without having to invoke the program under test.

After illustrating ISLa by example (Section 2), this paper makes the following contributions:

A specification language for input constraints. We propose a formalism (ISLa) for augmenting existing context-free grammars with context-sensitive constraints. We formally define the syntax and semantics of ISLa constraints (Section 3). ISLa has a rich *declarative* layer, separating semantic properties (constraints) from syntactic properties (the grammar).

Satisfying input constraints. We describe an efficient procedure to generate inputs satisfying ISLa constraints (and their reference grammars), prove its soundness and completeness (Section 4), and discuss our implementation (Section 5).

[2]Note that "isla" is also the Spanish word for *island*.

[3]https://github.com/google/AFL

A precise input generator. We evaluate ISLa by formalizing semantic properties from languages of quite different character, namely XML, a subset of C, reStructuredText, CSV files, and tar archives. Our results demonstrate that already a few lines of ISLa specifications suffice to *efficiently* generate *100% precise* inputs while maintaining *diversity*. To the best of our knowledge, this makes ISLa the first formalism for *checking and generating from* context-sensitive constraints of system inputs.

After discussing related work (Section 7), we close with conclusion and future work (Section 8).

2 ISLA BY EXAMPLE

Let us start by demonstrating how to formulate ISLa constraints along semantic properties of the XML language.[4] When randomly generating inputs from the XML grammar in Fig. 1 using a grammar fuzzer and passing those to an XML processor (e.g., Python's `xml.etree` package), we obtain not only one, but *three* kinds of errors: (1) "Mismatched tag", (2) "duplicate attribute", and (3) "unbound prefix." By adding ISLa specifications to the XML grammar, we can substantially increase the portion of valid XML we pass to an XML processor. Moreover, these specifications document XML features relevant for the parser of our test target.

Since ISLa is closed under conjunction, we do not need to come up with a monolithic specification of the semantic properties of valid XML at once. Rather, we can *incrementally* refine the specification simply by adding specifications of individual input properties until we are satisfied with the quality of the generated inputs or the value of the specification as a documentation measure.

From the inputs generated from the XML grammar, about 50% are invalid due to an unbound prefix, about 20% because of a mismatched tag, and roughly 3% due to a duplicate attribute. In the introduction, we already addressed "mismatched tag," which is easiest to specify. Next, we specify a property avoiding "unbound prefix" errors.

An "unbound prefix" error is raised when tag or attribute identifiers in XML documents contain a *namespace prefix*, such as `ns1` and `ns2` in `<ns1:tag ns2:attr="..."/>`, which is not *declared* in a surrounding tag. This is an example of a *def-use* property which is also common in programming languages: A *used* identifier must be *defined* in some outer scope or at some preceding position. To declare namespace `ns1`, one adds the attribute `xmlns:ns1="some_text"` to a surrounding tag, where usually (but not necessarily), the quoted text contains a URL. The property we aim for is expressed more precisely as: "*For all* identifiers with a prefix *p in* any XML tree, *there is* a *surrounding* XML tree *t* such that *there is* an attribute `xmlns:`*p in* the attributes list of *t*'s opening tag." We emphasized words corresponding to ISLa language elements. There is one subtlety, though: We have to distinguish prefixes in attribute and tag identifiers, since the special attribute `xmlns` does *not* have to be declared, as it is used precisely to declare other namespaces.

[4] We provide some additional examples in the appendix.

Listing 2: ISLa constraint for binding prefixes in attribute identifiers (reference grammar: Fig. 1)

```
1  const start: <start>;
2  vars {
3    prefix_id: <id-with-prefix>;
4    prefix_use, prefix_def: <id-no-prefix>;
5    outer_tag: <xml-tree>;
6    attribute, cont_attr, def_attr <xml-attribute>;
7    contained_tree: <inner-xml-tree>;
8  }
9  constraint {
10   forall attribute in start:
11     forall prefix_id="{prefix_use}:<id-no-prefix>" in
         ↪ attribute:
12       ((= prefix_use "xmlns") or
13         exists outer_tag="<<id> {cont_attr}>{contained_tree}
             ↪ </<id>>" in start:
14           (inside(attr, contained_tree) and
15            exists def_attr="xmlns:{prefix_def}=\"<text>\"" in
                 ↪  cont_attr:
16              (= prefix_use prefix_def)))
17 }
```

Again, we can express both cases in isolation to incrementally refine the specification. Here, we regard the slightly more complicated case of prefixes in attribute identifiers. Listing 2 shows the ISLa specification for this case. The ISLa code quite literally follows the natural language specification we described previously, except that we specialized it to only quantify over *attributes* (Line 10) and generally permit the `xmlns` prefix (Line 12) using a *disjunction*: Either the prefix is `xmlns`, *or* it must be explicitly defined.

With ISLa specifications, we can go beyond constraints for semantic validity for *application-specific, targeted testing*. Imagine an XML processor which allows associating tags with URLs that are defined using dedicated attributes `web:baseurl` and `web:query` for base URLs and query strings. We can enforce the existence of a tag using both of these attributes somewhere in any produced system input:

```
constraint {
  exists tree="<<id> {attributes}[/]>[<inner-xml-tree><xml-
      ↪ close-tag>]" in start:
    (exists attribute="{attr_id}=<text>"
       in attributes:
     (= attr_id "web:baseurl") and
     exists attribute="{attr_id}=<text>"
       in attributes:
```

```
    (= attr_id "web:query"))
}
```

The XML processor performs some input validation and rejects all inputs where the values of these attributes exceed a length of 100 characters. We force all generated inputs to respect this constraint by adding the following specification:

```
constraint {
  forall attribute="web:<id-no-prefix>={text}"
     in start:
    (<= (str.len text) 100)
}
```

After parsing an XML file, the processor assembles a complete URL by joining the base URL and the query string. However, let us assume its input validation is buggy: The result is stored in a character array of length 150, and we thus get a *buffer overflow* when the base URL and the query string *together* exceed a length of 150 characters. We can then *explicitly generate* inputs triggering this bug by encoding this property as an ISLa constraint. Such inputs would be valuable for attackers, targeting specific bugs, as well as for defenders, as a regression test validating a fix:

```
constraint {
  forall attributes in start:
    forall attribute_1="web:baseurl={text_1}"
       in attributes:
      forall attribute_2="web:query={text_2}"
         in attributes:
        (> (+ (str.len text_1) (str.len text_2))
           150)
}
```

With these examples, we have demonstrated how ISLa constraints precisely characterize input classes associated to some program behavior. Such descriptions can then be used to obtain *semantically valid* inputs, to *describe* the conditions of discovered bugs, or for targeted *triggering* of such bugs.

Note that implementing any of these constraints *without* ISLa can be a tiresome experience. While a handwritten generator can easily ensure matching XML tags or usage of tags from a dictionary, proper handling of namespaces is already a challenge, and solving arithmetic constraints over multiple elements will be increasingly difficult. Extending such a generator to be composable and also usable as a parser for checking or mutating inputs will require an effort comparable to implementing most of ISLa, but the resulting tool will not be nearly as versatile.

3 ISLA SYNTAX AND SEMANTICS

We formalize the syntax of the ISLa language. ISLa constraints are built from a *signature* of grammar, predicate, and variable symbols. We first formally define CFGs, following [13, Chapter 5]; afterward, we introduce ISLa signatures.

Definition 3.1 (Context-Free Grammar). A Context-Free Grammar (CFG) is a tuple $G = (N, T, P, S)$ of (1) a set of *nonterminal symbols* N, (2) a set of *terminal symbols* T disjoint from N, where the special terminal symbol $\epsilon \in T$ represents the empty string, (3) a set of *productions* P mapping nonterminal symbols $n \in N$ an *(expansion) alternative*. An alternative is a string of terminal or nonterminal symbols. Formally, $P \subseteq N \times (N \cup T)^k$ for some number $k > 0$; and (4) a designated *start symbol* $S \in N$.

By convention, we surround nonterminal symbols with angular brackets (e.g., $\langle start \rangle$). Signatures contain a special nonterminal symbol NUM which we use for *numeric* constants: Symbols of this type represent derivation trees whose string representations correspond to a natural number.

Definition 3.2 (ISLa Signature). A *signature* is a tuple $\Sigma = (G, \text{PSym}, \text{VSym})$ of a *grammar* $G = (N, T, P, S)$, a set of *predicate symbols* PSym of strictly positive *arity*, and a set of typed variable symbols VSym. The type $vtype(v)$ of $v \in \text{VSym}$ is a symbol $n \in N \cup \{\text{NUM}\}$, $\text{NUM} \notin N$.

In the following definition of ISLa formulas, we assume an underspecified set $\text{Trm}_{bool}(\text{VSym})$ of SMT-LIB formulas (terms of sort *Bool*). That is, this set contains the constants `true` and `false`, and S-expressions `(f` a_1 `...` a_n`)`, where `f` is an n-ary function symbol of *Bool* sort and the a_i are SMT expressions of suitable sort. Formulas in $\text{Trm}_{bool}(\text{VSym})$ may contain uninterpreted String constants whose names coincide with the names in VSym. For the precise definition of SMT-LIB terms, we refer to the SMT-LIB standard [1] and the repository of SMT-LIB theories [24].

Definition 3.3 (ISLa Formulas). The set Fml of *ISLa formulas* for a signature $\Sigma = (G, \text{PSym}, \text{VSym})$, with $G = (N, T, P, S)$, is inductively defined as:

(1) $\varphi \in \text{Fml}$ if $\varphi \in \text{Trm}_{bool}(\text{VSym})$.
(2) $p(v_1, \ldots, v_n) \in \text{Fml}$ each predicate symbol $p \in \text{PSym}$ with arity n and $v_i \in \text{VSym}$.
(3) $(\textbf{not}\ \varphi)$, $(\varphi\ \textbf{and}\ \psi)$, $(\varphi\ \textbf{or}\ \psi)$ are in Fml for $\varphi, \psi \in \text{Fml}$.
(4) **forall** x **in** y: φ and **exists** x **in** y: φ are in Fml for $x, y \in \text{VSym}$ and $\varphi \in \text{Fml}$.
(5) For $x, y \in \text{VSym}$, $\varphi \in \text{Fml}$, **forall** x="*mexp*" **in** y: φ and **exists** x="*mexp*" **in** y: φ are in Fml, where *mexp* is a string consisting of symbols in $N \cup T$, of non-nested lists of symbols in $N \cup T$ (*optional* symbols), and of variables $v \in \text{VSym}$.
(6) **num** n: φ is in Fml for $n \in \text{VSym}$ and $\varphi \in \text{Fml}$.

The set Fml is always relative to a signature Σ, which we leave implicit in the notation for simplicity (instead of, e.g., writing Fml_Σ). We also allow omitting parentheses in accordance with the following precedence rules: Quantifiers

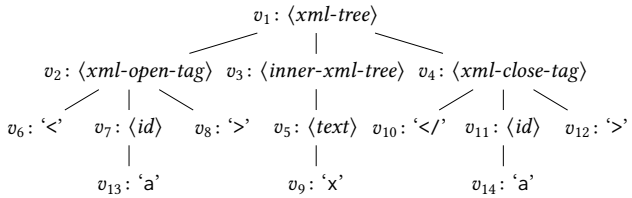

Figure 2: Example XML derivation tree.

and **num** bind stronger than negation, which binds stronger than conjunction, which binds stronger than disjunction.

Let, for $\varphi \in \text{Fml}$, $fv(\varphi)$ denote the set of *free variables* in φ, i.e., variables that are not bound by quantifiers or match expressions (for SMT-LIB S-expressions, this is the set of uninterpreted constants of String sort). In the following, we only consider (top-level) ISLa formulas which contain *exactly one* free variable, which is the one declared in the **const** section in the concrete syntax.

3.1 Semantics

The semantics of an ISLa constraint are all strings that can be derived from the reference grammar which satisfy the constraint. To formalize this intuition, we first define derivation trees and the language of CFGs. Then, we fix the meaning of ISLa constraints by defining a validation judgment.

In the following definition of derivation trees, we use the symbols $<$ and $\leq$ to denote the strict and non-strict versions of the same partial order relation, respectively; for the corresponding covering relation which only holds between parents and their *immediate* children, we write $\prec$.

Definition 3.4 (Derivation Tree). A *derivation tree* for a CFG $G = (N, T, P, S)$ is a *rooted ordered tree* $t = (X, \leq_V, \leq_S)$ such that (1) the vertices $v \in X$ are *labeled* with symbols $label(v) \in N \cup T$, (2) the *vertical* order $\leq_V \subseteq X \times X$ indicates the parent-child relation such that the partial order $(X, \leq_V)$ forms an unordered tree, (3) the *sibling* order $\leq_S \subseteq X \times X$ yields a partial order $(X, \leq_S)$ such that two distinct nodes v_1, v_2 are comparable by relation $\prec_S$ if, and only if, they are siblings, (4) the root of t is labeled with S, and (5) each inner node v is labeled by a symbol in $n \in N$ and, if $v_1, \ldots, v_k$ is the ordered list of all immediate children of v, i.e., all distinct nodes such that $v \prec_V v_i$ and $v_i \prec_S v_l$ for $1 \leq i < l \leq k$, there is a production $(n, s_1, \ldots, s_k) \in P$ such that $label(v) = n$ and, for all v_i, $label(v_i) = s_i$. We write $leaves(t)$ for the set of leaves of t, and $label(t)$ for the label of its root. A derivation tree is *closed* if $l \in T$ for all $l \in leaves(t)$, and *open* otherwise. $\mathcal{T}(G)$ is the set of all (closed and open) derivation trees for G.

Example 3.5. Fig. 2 visualizes the derivation tree for the XML document <a>x. Formally, this tree is represented as a triple $t = (X, \leq_V, \leq_S)$, where $X = \{v_1, \ldots, v_{14}\}$, with $label(v_1) = \langle xml\text{-}tree \rangle$, $label(v_2) = \langle xml\text{-}open\text{-}tag \rangle$, etc. The *vertical order* $\leq_V$ contains the edges in the figure: For example, $v_1 \leq_V v_2$ and $v_2 \leq_V v_{13}$. This relation alone only gives us an unordered tree: When "unparsing" the tree, we could thus obtain the undesired result xa><></a. Thus, we define a *sibling order* $\leq_S$ to order the immediate children of the same node. For instance, we have $v_6 \leq_S v_8$ (and $v_6 \leq_S v_6$, $v_6 <_S v_8$, and $v_6 <_S v_7$), but not $v_6 \leq_S v_9$ (since they have different parents) and $v_6 \prec_S v_8$ (since they are not *immediate* siblings). The tree is not only any ordered tree, but a *derivation tree* for the XML grammar in Fig. 1, since it satisfies Items (4) and (5) of Definition 3.4. The root of the tree, v_1, is labeled with the grammar's start symbol $\langle xml\text{-}tree \rangle$ (Item (4)). The tree relations conform to the possible grammar derivations: Consider, e.g., node v_2 and its immediate children v_6, v_7, and v_8. According to Item (5), there has to be a production $(\langle xml\text{-}open\text{-}tag \rangle, \text{'<'}, \langle id \rangle, \text{'>'})$ in the grammar, which is indeed the case, since '<' $\langle id \rangle$ '>' is an expansion alternative (the first one) for the nonterminal $\langle xml\text{-}open\text{-}tag \rangle$. The leaf set $leaves(t)$ is $\{v_6, v_{13}, v_8, v_9, v_{10}, v_{14}, v_{12}\}$. The tree t is *closed*, since all leaves are labeled with *terminal* symbols. It would be *open* if we removed the subtree rooted in any tree node (but the root; then there would be no tree any more).

To convert derivation trees to strings, we define a *global tree order* between tree nodes. The interesting property of this relation is that it provides a strict partial order on the leaf set of a derivation tree; thus, we can chain an ordered leaf set to a string representation of a tree.

Definition 3.6 (Global Tree Order). Let $t = (X, \leq_V, \leq_S)$ be an ordered tree. The *global tree order* $\ll$ is defined as $v_1 \ll v_2$ if, and only if, it holds that and there are two parent nodes p_1, p_2 (not necessarily different from v_1, v_2) such that (1) $p_1 \leq_V v_1$ and $p_2 \leq_V v_2$, (2) $p_1 \prec_S p_2$, and (3) there are no nodes $p', s' \in X$ such that (3.1) $p' \leq_V v_1$, $p_1 <_V p'$, and $p' <_S s'$, *or* (3.2) $p' \leq_V v_2$, $p_2 <_V p'$, and $s' <_S p'$.

In Fig. 2, it holds that $v_2 \ll v3$ since they are immediate siblings themselves ($p_1 = v_2$, $p_2 = v_3$), and $v_8 \ll v_5$ since they have parents v_2 and v_3 that are immediate siblings; but not, e.g., $v_7 \ll v_9$ since $p' = v_7$ has a greater sibling $s' = v_8$, and not $v_2 \ll v_4$ since they are non-immediate siblings. Furthermore, $v_9 \not\ll v_{14}$, since v_{14} has an intermediate parent v_{11} that has a smaller sibling (v_{10}). Note that all pairs of leaves in the order in which they occur in a traversal of the tree are in $\ll$, e.g., $v_6 \ll v_{13}$ and $v_9 \ll v_{10}$.

Definition 3.7 (Tree to String Conversion). Let $t \in \mathcal{T}(G)$ be a derivation tree, and $l_1, \ldots, l_n$ be all leaves of t, such that $l_i \ll l_m$ for $1 \leq i < m \leq n$. Then $str(t) := (label(l_1), \ldots, label(l_n))$.

For t being the tree from Fig. 2, we have

$$str(t) = (\text{`<'}, \text{`a'}, \text{`>'}, \text{`x'}, \text{`</'}, \text{`a'}, \text{`>'}).$$

Matching Match Expressions. For evaluating ISLa formulas, we have to match quantified formulas with match expressions against derivation trees. To that end, we use a function $match(t, mexpr)$ which is applies to trees t and match expressions $mexpr$. Its return type is mapping of variables to subtrees. We say that *there is a match* m for t and $mexpr$ if $match$ returns a non-empty mapping. Our implementation performs a greedy match, but uses backtracking to account for, e.g., recursive structures. If there are optional elements in the match expression, it considers all possibilities of present and non-present optionals. We provide more details in the appendix.

Validation. The semantics of ISLa formulas is defined by a *validation judgment* $\pi, \sigma, \beta \models \varphi$, where π and σ are interpretations of predicate symbols and SMT expressions, and the variable assignment β is a substitution of derivation trees for variables. The intuition of this judgment is that φ holds (evaluates to *true*) when assigning free variables in φ according to β under the interpretations of predicates and SMT expressions as provided by π and σ. We write $\beta(\varphi)$ for the substitution of free variables in φ by their assignments in β, and $\beta[v \mapsto t]$ for the updated assignment where the variable v is now mapped to the tree t. For a match $m = match(t, mexpr,)$, we write $\beta[m]$ for $\beta[v_1 \mapsto t_1] \cdots [v_n \mapsto t_n]$, where $v_i \mapsto t_i$ are all assignments in m. The primitive substitution of t for v is denoted by $\{v \mapsto t\}$. For any β, we denote by $\beta^{\downarrow}$ the assignment of variables to *strings* instead of trees: If β associates v with t, $\beta^{\downarrow}$ associates v with $str(t)$.

In the definition of the validation judgment, we use $\top$ to represent semantic truth, and $\bot$ for falsity. Note that we expect σ to always return $\top$ or $\bot$. While timeouts are common for SMT solvers, they usually do not occur for quantifier-free formulas with concrete values (i.e., no uninterpreted constants). In ISLa specifications, we commonly abstain from quantifiers at the SMT level (and only use ISLa quantifiers). Should the solver time out anyway, we interpret this as $\bot$.

Definition 3.8 (ISLa Validation). Let $\Sigma = (G, \text{PSym}, \text{VSym})$ be a signature, $\pi : \text{PSym} \to \mathcal{T}(G)^* \to \{\top, \bot\}$ an interpretation of predicate symbols, $\sigma : \text{Trm}_{bool}(\emptyset) \to \{\top, \bot\}$ an interpretation of closed SMT S-expressions, and β a variable assignment. We inductively define the judgment $\pi, \sigma, \beta \models \varphi$ as

(1) $\pi, \sigma, \beta \models \varphi$ iff $\varphi \in \text{Trm}_{bool}(\emptyset)$ and $\sigma(\beta^{\downarrow}(\varphi)) = \top$.
(2) $\pi, \sigma, \beta \models p(v_1, \ldots, v_n)$ iff $\pi(p)(\beta(v_1), \ldots, \beta(v_n)) = \top$.
(3) $\pi, \sigma, \beta \models$ **not** φ iff not $\pi, \sigma, \beta \models \varphi$.
(4) $\pi, \sigma, \beta \models \varphi$ **and** ψ iff $\pi, \sigma, \beta \models \varphi$ and $\pi, \sigma, \beta \models \psi$.
(5) $\pi, \sigma, \beta \models \varphi$ **or** ψ iff $\pi, \sigma, \beta \models \varphi$ or $\pi, \sigma, \beta \models \psi$.
(6) $\pi, \sigma, \beta \models$ **forall** v **in** w: φ iff $\pi, \sigma, \beta[v \mapsto t] \models \varphi$ holds for all subtrees t in $\beta(w)$ whose root is labeled with $vtype(v)$.
(7) $\pi, \sigma, \beta \models$ **exists** v **in** w: φ iff $\pi, \sigma, \beta[v \mapsto t] \models \varphi$ holds for some subtree t in $\beta(w)$ whose root is labeled with $vtype(v)$.
(8) $\pi, \sigma, \beta \models$ **forall** v="$mexpr$" **in** w: φ iff $\pi, \sigma, \beta[v \mapsto t][m] \models \varphi$ holds for all subtrees t in $\beta(w)$ whose root is labeled with $vtype(v)$ and for which there is a match $m = match(t, mexpr)$.
(9) $\pi, \sigma, \beta \models$ **exists** v="$mexpr$" **in** w: φ iff $\pi, \sigma, \beta[v \mapsto t][m] \models \varphi$ holds for some subtree t in $\beta(w)$ whose root is labeled with $vtype(v)$ and for which there is a match $m = match(t, mexpr)$.
(10) $\pi, \sigma, \beta \models$ **num** n: φ iff $\pi, \sigma, \beta[n \mapsto t] \models \varphi$ holds for some tree t such that $str(t)$ represents a natural number (naturally including zero).

Example 3.9. Consider the constraint for well-balanced XML trees from Listing 1, and the XML tree t from Fig. 2 for the document `<a>x</a>`. We evaluate whether this tree is well-formed, starting from an initial assignment $\beta = \{\texttt{start} \mapsto t\}$. Since the outermost element of the constraint is a universal formula with match expression, Item (8) of Definition 3.8 applies. Thus, we have to prove that $\pi, \sigma, \{\texttt{start} \mapsto t\}[v \mapsto t][m] \models \varphi$ holds for all instantiations, i.e., tree elements with root $\langle xml\text{-}tree \rangle$ matching the match expression (i.e., not a self-closing tag). There is exactly one such match m in t, instantiating `opid` to `a` and `clid` to `a`. Thus, it remains to show that $\sigma(\texttt{(= "a" "a")}) = \top$, which is indeed the case.

Definition 3.10 (ISLa Semantics). Let $\varphi \in \text{Fml}$ be an ISLa formula with the single free variable c for the signature $\Sigma = (G, \text{PSym}, \text{VSym})$, and π, σ be interpretations for predicates and SMT formulas. We define the semantics $[\![\varphi]\!]$ of φ as

$$[\![\varphi]\!] := \{str(t) \mid t \in \mathcal{T}(G) \wedge leaves(t) = \emptyset \wedge \pi, \sigma, \{c \mapsto t\} \models \varphi\}.$$

4 INPUT GENERATION

We formalize the input generation for ISLa as a transition system between *Conditioned Derivation Trees (CDTs)* $\Phi \triangleright t$, where Φ is a set of ISLa formulas (interpreted as a conjunction) and t a (possibly open) derivation tree. Intuitively, $\bigwedge \Phi$ constrains the inputs represented by t, similarly as $[\![\varphi]\!]$ constrains the language of the grammar. To make this possible, we need to relax the definition of ISLa formulas: Instead of free variables, formulas may contain references to tree nodes which they are concerned about. To that end, tree nodes are assigned unique, numeric identifiers, which may occur everywhere in ISLa formulas where a free variable might

occur (variables bound by quantifiers may not be replaced with tree identifiers).

Consider, for example, the ISLa constraint

$$\varphi = \textbf{forall } id \textbf{ in } start\colon \texttt{(= (str.len } id\texttt{) 17)}$$

constraining the XML grammar in Fig. 1 to identifiers of length 17, where *id* is a bound variable of type ⟨*ID*⟩ and *start* is a free variable of type ⟨*start*⟩. Let t be a tree consisting of a single (root) node with identifier 1, and labeled with ⟨*start*⟩. Then, $[\![\varphi]\!]$ is identical to the strings represented by the CDT

$$\{\textbf{forall } id \textbf{ in } 1\colon \texttt{(= (str.len } id\texttt{) 17)}\} \triangleright t.$$

Our CDT transition system relates an input CDT to a set of output CDTs. We define two properties of such transitions: A transition is *precise* if the input represents *at most* the set of all strings represented by all outputs together; conversely, it is *complete* if the input represents *at least* the set of all strings represented by all outputs. Precision is mandatory for the ISLa producer, since we have to avoid generating system inputs which do not satisfy the specified constraints.

To define the semantics of CDTs, we first define the closed trees represented by (the language of) *open* derivation trees. We need the concept of a *tree substitution*: The tree $t[v \mapsto t']$ results from $t = (X, \leq_V, \leq_S)$ by replacing the subtree rooted in node $v \in X$ by t', updating X, $\leq_V$ and $\leq_S$ accordingly.

Definition 4.1 (Semantics of Open Derivation Trees). Let $t \in \mathcal{T}(G)$ be a derivation tree for a grammar $G = (N, T, P, S)$. We define the set $\mathcal{T}(t) \subseteq \mathcal{T}(G)$ of closed derivation trees represented by t as

$$\begin{aligned}\mathcal{T}(t) \coloneqq \big\{ t[l_1 \mapsto t_1] \cdots [l_k \mapsto t_k] \mid \\ & l_i \in \mathit{leaves}(t) \wedge k = |\mathit{leaves}(t)| \\ & \wedge\ (\forall j, m \in 1 \ldots k : l \neq m \rightarrow l_j \neq l_m) \\ & \wedge\ n_i = \mathit{label}(t_i) = \mathit{label}(l_i) \\ & \wedge\ G_{n_i} = (N, T, P, n_i) \wedge t_i \in \mathcal{T}(G_{n_i}) \big\}\end{aligned}$$

Observe that for the tree consisting of a single node labeled with the start symbol S, $\mathcal{T}(t)$ is identical to $\mathcal{T}(S)$. Furthermore, for any *closed* tree t', it holds that $\mathcal{T}(t') = \{t'\}$.

We re-use the validity judgment defined from Definition 3.8 for the semantics definition for CDTs by interpreting tree identifiers in formulas similarly to variables. Furthermore, the special variable assignment β_t for the derivation tree t associates with each tree identifier in t the subtree rooted in the node with that identifier. Then, the definition is a straightforward specialization of Definition 3.10:

Definition 4.2 (Semantics of CDTs). Let $\Phi \subseteq 2^{\text{Fml}}$ be a set of ISLa formulas for the signature $\Sigma = (G, \text{PSym}, \text{VSym})$, $t \in \mathcal{T}(G)$ be a derivation tree, and π, σ be interpretations for predicates and SMT formulas. We define the semantics $[\![\Phi \triangleright t]\!]$ of the CDT $\Phi \triangleright t$ as

$$[\![\Phi \triangleright t]\!] \coloneqq \{\mathit{str}(t') \mid t' \in \mathcal{T}(t) \wedge \mathit{leaves}(t') = \emptyset \wedge \pi, \sigma, \beta_{t'} \models \bigwedge \Phi\}.$$

A CDT Transition System (CDTTS) is simply a transition system between CDTs.

Definition 4.3 (CDT Transition System). A *CDTTS* for a signature $\Sigma = (G, \text{PSym}, \text{VSym})$ is a transition system $(C, \rightarrow)$, where, for $\Phi \in 2^{\text{Fml}}$ and $t \in \mathcal{T}(G)$, C consists of CDTs $\Phi \triangleright t$, and $\rightarrow \subseteq C \times C$. We write $cdt \rightarrow cdt'$ if $(cdt, cdt') \in' \rightarrow$.

Intuitively, one applies CDTTS transitions to an initial constraint with a trivial tree only consisting of a root node labeled with the start nonterminal, and collects "output" CDTs $\emptyset \triangleright t$ with empty constraint. The trees t of such outputs are solutions to the initial problem. We call a CDTTS *globally precise* if all such trees t are actual solutions, i.e., the system does not produce wrong outputs; we call it *globally complete* if the entirety of trees t from result CDTs represents the full semantics of the input CDT.

Definition 4.4 (Global Precision and Completeness). Let $(C, \rightarrow)$ be a CDTTS, and R_{cdt} be the set of all closed trees t such that $cdt \rightarrow \cdots \rightarrow \emptyset \triangleright t$ is a derivation in $(C, \rightarrow)$. Then, $(C, \rightarrow)$ is *globally precise* if, for each CDT cdt in the domain of $\rightarrow$, it holds that $[\![cdt]\!] \supseteq \{\mathit{str}(t) | t \in R_{cdt}\}$. The CDTTS is *globally complete* if it holds that $[\![cdt]\!] \subseteq \{\mathit{str}(t) | t \in R_{cdt}\}$.

To enable transition-local reasoning about precision and completeness, we define notions of local precision and completeness. Local precision is the property that *at each transition step*, no "wrong" inputs are added, and local completeness the property that no transition step loses information.

Definition 4.5 (Local Precision and Completeness). A CDTTS $(C, \rightarrow)$ is *precise* if, for each CDT cdt in the domain of $\rightarrow$, it holds that $[\![cdt]\!] \supseteq \bigcup_{cdt \rightarrow cdt'} ([\![cdt']\!])$. The CDTTS is *complete* if it holds that $[\![cdt]\!] \subseteq \bigcup_{cdt \rightarrow cdt'} ([\![cdt']\!])$.

As for "soundness" in first-order logic (see, e.g., [26]), local precision implies global precision, i.e., it suffices to show that the individual transitions are precise to obtain the property for the whole system. This is demonstrated by the following Lemma 4.6. Note that the opposite direction does not hold, since a CDTTS could in theory lose precision locally and recover it globally, although it is unclear how (and why) such a system should be designed.

Lemma 4.6. *A locally precise CDTTS is also globally precise.*

Proof. The lemma trivially holds if $R_{cdt} = \emptyset$. Otherwise, let $cdt_0 \longrightarrow cdt_1 \longrightarrow \cdots \longrightarrow \emptyset \triangleright t$ be any transition chain s.t. $cdt_0 = cdt$ and $t \in R_{cdt}$. Then, it follows from local precision that $[\![cdt_k]\!] \supseteq [\![cdt_{k+1}]\!]$ for $k = 0, \ldots, n-1$, and by transitivity of $\supseteq$ also $[\![cdt_k]\!] \supseteq [\![cdt_l]\!]$ for $0 \leq k < l \leq n$. Since $[\![\emptyset \triangleright t']\!] = str(t')$ for closed t', the lemma follows. □

Global completeness cannot easily be reduced to local completeness: It includes the "termination" property that all derivations end in CDTs with empty constraint set; furthermore, one has to show that there is an applicable transition for each CDT with non-empty semantic.

Our ISLa solver prototype implements the CDTTS in Fig. 3. It solves SMT and semantic predicate constraints by querying the SMT solver or the predicate oracle, and eliminates existential constraints by inserting new subtrees into the current conditioned tree. Only when the complete constraint has been eliminated, we finish off the remaining incomplete tree by replacing open leaves with suitable concrete subtrees. This is in principle a complete procedure; yet, our implementation (cf. Section 5) only considers a finite subset of all solutions in solver queries and when performing tree insertion. Consequently, it usually misses some solutions, but outputs *more diverse results more quickly* compared, e.g., to a naive search-based approach filtering out wrong solutions.

Transition Rules. In the ISLa CDTTS, we use *indexed* CDTs $\Phi \triangleright^I t$. In the index set I, we track previous matches of universal quantifiers to make sure that we do not match the same trees over and over. Since SMT formulas can now also contain variables, evaluating them can result in a *model* β (an assignment). Note that we can obtain different assignments by repeated solver calls (negating previous solutions). We divide the set PSym of predicate symbols into two disjoint sets PSym_{st} and PSym_{sem} of *structural* and *semantic* predicates. Structural predicates address constraints such as *before* or *within*, and they evaluate to $\top$ or $\bot$. *Semantic* predicates formalize constraints such as specific checksum implementations. They may additionally evaluate to a set of assignments, as in the case of satisfiable SMT expressions, or to the special value "not ready" (denoted by $\circlearrowleft$). Intuitively, an evaluation results in $\top$ ($\bot$) if all of (not any of) the derivation trees represented by an abstract tree satisfy the predicate. Assignments is returned if the given tree can be completed or "fixed" to a satisfying solution. One may obtain $\circlearrowleft$ if the constrained tree lacks sufficient information for such a computation (e.g., the inputs of a checksum predicate are not yet determined).

We explain the individual transition rules of the CDTTS from Fig. 3. Rule (1) uses a function $inv : \text{Fml} \longrightarrow 2^{\text{Fml}}$ to enforce the invariant that all formulas $\varphi \in \Phi$ are in Negation Normal Form (NNF) and do not contain top-level conjunctions and disjunctions. Basically, *inv* converts its input into Disjunctive Normal Form and returns the disjunctive elements. It is only applicable to CDTs whose constraints do not satisfy the invariant. Rule (2) eliminates satisfied structural predicate formulas from a constraint set. Numeric constant introductions are eliminated in Rule (3) by introducing a fresh (not occurring in the containing CDT) variable symbol with the special nonterminal type NUM for natural numbers.

Rules (4) and (5) eliminate universal formulas that have already been matched with all applicable subtrees, and which cannot possibly be matched against *any* extension of the (open) tree. This is the case if the nonterminal type of the quantified variable is not reachable from any leaf and, if there is a match expression, the current tree cannot be completed to a matching one.

Universal formulas with and without match expressions are subject of Rules (6) and (7). First, matching subtrees of the tree $\beta_t(id)$ identified with id are collected in a set T. We only consider subtrees that are not already matched, i.e., where (ψ, t') is not yet in the index set I. If T is empty, the rules are not applicable. Otherwise, the set Φ of all instantiations of φ according to the discovered matches is added to the constraint set. We record the instantiations (ψ, t'), for all matched trees t', in the index set. The output of these rules is a singleton.

If universal quantifiers remain which cannot be matched or eliminated, we expand the current tree in Rule (8). The function $expand^{\forall}_{\Phi}(t)$ returns all possible trees t' in which each open leaf has been expanded *one step* according to the grammar. However, we only expand leaves which are bound by a universal quantifier, that is, which represent possible subtrees that could be unified with a universally quantified formula. For this reason, we pass Φ as an argument. We call the remaining, unbound grammar symbols *free nonterminals*. For example, the XML constraint in Listing 1 does not restrict the instantiation of $\langle text \rangle$ nonterminals. Thus, $\langle text \rangle$ is a free nonterminal which we will not expand with Rule (8). Instead, such nonterminals are instantiated to concrete closed subtrees in a single step by Rules (15) and (16). In our implementation, we use a standard coverage-based fuzzer to that end. Thus, we avoid producing many strings which only differ, e.g., in identifier names or text passages within XML tags.

Rules (9) and (10) eliminate *satisfiable* SMT or semantic predicate formulas by querying σ or π (there is no transition for unsatisfiable or "not ready" formulas). The transition result consists of one instantiation per returned assignment β.

The only remaining constraints—in satisfiable constraint sets— are existential formulas, and semantic predicate formulas that are not yet ready to provide a solution. Existential formulas can be matched just like universal ones; but instead

$$\{\dots,\varphi,\dots\} \triangleright^I t \;\rightarrow\; \{\{\dots,\varphi'\dots\} \triangleright^I t \mid \varphi' \in inv(\varphi) \wedge \varphi \neq \varphi'\} \neq \emptyset \tag{1}$$

$$\Phi \triangleright^I t \;\rightarrow\; \{\Phi \setminus \varphi \triangleright^I t \mid \varphi \in \Phi \cap \mathrm{PSym}_{st} \wedge\ \pi(\varphi) = \top\} \tag{2}$$

$$\{\dots,\texttt{num}\ n\colon \varphi,\dots\} \triangleright^I t \;\rightarrow\; \{\dots,\{n \mapsto c\}(\varphi),\dots\} \triangleright^I t \tag{3}$$
where $c \in \mathrm{VSym}$ is *fresh* and $vtype(c) = \mathsf{NUM}$

$$\{\dots,\overbrace{\texttt{forall}\ v\ \texttt{in}\ id\colon \varphi}^{\psi},\dots\} \triangleright^I t \;\rightarrow\; \{\{\dots,\dots\} \triangleright^I t\} \tag{4}$$
if $\forall$ subrees t' of $\mathcal{T}(\beta_t(id))$:
$(\psi, t') \in I \vee label(t') \neq vtype(v)$

$$\{\dots,\overbrace{\texttt{forall}\ v{=}\text{“}mexpr\text{”}\ \texttt{in}\ id\colon \varphi}^{\psi},\dots\} \triangleright^I t \;\rightarrow \tag{5}$$
$\{\{\dots,\dots\} \triangleright^I t\}$ if $\forall$ subrees t' of $\mathcal{T}(\beta_t(id))$:
$(\psi, t') \in I \vee label(t') \neq vtype(v) \vee$
there is no $m = match(t', mexpr)$

$$\{\dots,\overbrace{\texttt{forall}\ v\ \texttt{in}\ id\colon \varphi}^{\psi},\dots\} \triangleright^I t \;\rightarrow \tag{6}$$
$\left\{\{\dots,\psi,\dots\} \cup \bigcup \Psi \triangleright^{I \cup (\{\psi\}\times T)} t\right\}$ where
$\Psi = \left\{\beta_t[v \mapsto t'](\varphi) \mid t' \in T\right\} \wedge$
$T = \left\{t' \mid t' = \beta_t(id) \wedge (\psi, t') \notin I \wedge label(t') = vtype(v)\right\} \neq \emptyset$

$$\{\dots,\overbrace{\texttt{forall}\ v{=}\text{“}mexpr\text{”}\ \texttt{in}\ id\colon \varphi}^{\psi},\dots\} \triangleright^I t \;\rightarrow \tag{7}$$
$\left\{\{\dots,\psi,\dots\} \cup \bigcup \Psi \triangleright^{I \cup (\{\psi\}\times T)} t\right\}$ where
$\Psi = \left\{\beta_t[v \mapsto t'][m](\varphi) \mid (t', m) \in T\right\} \wedge$
$T = \left\{(t', m) \mid t' = \beta_t(id) \wedge (\psi, t') \notin I \wedge label(t') = vtype(v) \wedge \text{there is an } m = match(t, mexpr)\right\} \neq \emptyset$

$$\Phi \triangleright^I t \;\rightarrow\; \{\Phi \triangleright^I t' \mid t' \in expand^{\forall}_{\Phi}(t) \neq \emptyset\} \tag{8}$$

$$\Phi \triangleright^I t \;\rightarrow\; \{\Phi \setminus \varphi \triangleright^I \beta(t) \mid \varphi \in \Phi \cap \mathrm{Trm}_{bool}(\mathrm{VSym}) \wedge \beta \in \sigma(\varphi) \neq \bot\} \tag{9}$$

$$\Phi \triangleright^I t \;\rightarrow\; \{\Phi \setminus \varphi \triangleright^I \beta(t) \mid \varphi \text{ is } first\ \varphi \in \Phi \cap \mathrm{PSym}_{sem} \wedge \beta \in \pi(\varphi) \notin \{\bot, \circlearrowleft\}\} \tag{10}$$

$$\{\dots,\overbrace{\texttt{exists}\ v\ \texttt{in}\ id\colon \varphi}^{\psi},\dots\} \triangleright^I t \;\rightarrow \tag{11}$$
$\bigcup_{\xi\in\Psi} \left\{\{\dots,\xi,\dots\} \triangleright^I t\right\}$ where
$\Psi = \left\{\beta_t[v \mapsto t'](\varphi) \mid t' \in T\right\} \wedge$
$T = \left\{t' \mid t' = \beta_t(id) \wedge (\psi, t') \notin I \wedge label(t') = vtype(v)\right\} \neq \emptyset$

$$\{\dots,\overbrace{\texttt{exists}\ v{=}\text{“}mexpr\text{”}\ \texttt{in}\ id\colon \varphi}^{\psi},\dots\} \triangleright^I t \;\rightarrow \tag{12}$$
$\bigcup_{\xi\in\Psi} \left\{\{\dots,\xi,\dots\} \triangleright^I t\right\}$ where
$\Psi = \left\{\beta_t[v \mapsto t'][m](\varphi) \mid (t', m) \in T\right\} \wedge$
$T = \left\{(t', m) \mid t' = \beta_t(id) \wedge (\psi, t') \notin I \wedge label(t') = vtype(v) \wedge \text{there is an } m = match(t, mexpr)\right\} \neq \emptyset$

$$\{\dots,\texttt{exists}\ v\ \texttt{in}\ id\colon \varphi,\dots\} \triangleright^I t \;\rightarrow \tag{13}$$
$\left\{\{\dots,\{v \mapsto nid\}(\varphi),\dots\} \cup \Phi_{orig} \triangleright^I \{id \mapsto t'\}(t) \mid (nid, t') \in insert(makeTree(v), \beta_t(id))\right\}$

$$\{\dots,\texttt{exists}\ v{=}\text{“}mexpr\text{”}\ \texttt{in}\ id\colon \varphi,\dots\} \triangleright^I t \;\rightarrow \tag{14}$$
$\left\{\{\dots,\{v \mapsto nid\}(\varphi),\dots\} \cup \Phi_{orig} \triangleright^I \{id \mapsto t'\}(t) \mid (nid, t') \in insert(makeTree(v, mexpr), \beta_t(id))\right\}$

$$\emptyset \triangleright^I t \;\rightarrow\; \{\emptyset \triangleright^I t' \mid t' \in \mathcal{T}(t)\} \neq \{\emptyset \triangleright^I t\} \tag{15}$$

$$\Phi \triangleright^I t \;\rightarrow\; \{\Phi \triangleright^I t' \mid t' \in \mathcal{T}(t)\} \neq \{\Phi \triangleright^I t\} \tag{16}$$
if $\Phi \subseteq \{p(\dots) \mid p \in \mathrm{PSym}_{sem}\}$

Figure 3: Efficient ISLa CDTTS Transition Relation

of returning one result with all matches, Rules (11) and (12) return a *set* of solutions with one match each.

In addition to matching, we provide two rules Rules (13) and (14) to eliminate existential formulas using *tree insertion*. Note that, as exception to the general principle that the rules in the CDTTS are mutually exclusive, we can apply Rules (11) and (12) *and* Rules (13) and (14) wherever possible. The insertion routine *insert*(*newTree*, *t*,) guarantees that all returned results contain all nodes from the original tree t as well as the complete tree *newTree*. Nevertheless, tree insertion is an aggressive operation that may violate constraints that were satisfied before. For this reason, we have to add the original constraint Φ_{orig}, from which we started solving, again to the constraint set; if the tree insertion did not violate structural constraints, the original constraint can usually be quickly eliminated. However, tree insertion can also entail the necessity of subsequent tree insertions, e.g., if a new identifier was added that needs to be declared. Our implemented insertion

routine prioritizes structurally simple solutions, for which this is usually not necessary. In the appendix, we provide details on tree insertion.

Finally, Rules (15) and (16) "finish off" the remaining open derivation trees by replacing all open leaves with suitable concrete subtrees. In the case of Rule (15), this yields a decisive result of the CDTTS. Rule (16) addresses residual "not ready" semantic predicate formulas. We compute the represented closed subtrees such that all information for evaluating the semantic predicates is present. After this step, Rule (10) must be applicable.

In the appendix, we argue for the correctness of the subsequent precision and completeness theorems.

Theorem 4.7. *The ISLa CDTTS in Fig. 3 is globally precise.*

Theorem 4.8. *The ISLa CDTTS in Fig. 3 is globally complete.*

5 IMPLEMENTATION

Our ISLa solver implements the CDTTS in Fig. 3.[5] The system is implemented in about 4.4K lines of Python code. Basically, the solver maintains a priority queue of CDTs, selects the first element of the queue, and applies the *first applicable* transition (with the exception of Rules (11) to (14): We both match existential quantifiers *and* perform tree insertion whenever possible). Since transitions can result in multiple output states, we use a cost function to order elements in the queue. The cost function considers five cost factors like tree complexity (cost of finishing a tree) and coverage metrics; details are in the appendix. The cost function can be tuned by providing a *weight vector*; the total cost of a CDT is the weighted geometric average of the individual cost factors.

Three components are dedicated to solving individual constraints: We interface to Z3 for SMT formulas, use the individual solving routines of semantic predicates, and a tree insertion component for existential quantifiers. In addition, we developed a library for interpreting CFGs as graphs, which supports, e.g., distance computation, subgraph extraction, and filtering. Another library we implemented approximates the possible instantiations of a nonterminal in a grammar by regular expressions. This is used to constrain solutions of the SMT solver to values which are admissible in the grammar.

For instantiating unconstrained nonterminals in derivation trees (Rules (15) and (16) in the ISLa CDTTS), we use an existing grammar-based fuzzer which aims to maximize (grammar) coverage. The number of outputs which should be generated from such unconstrained trees is a configuration parameter of the solver. Another parameter controls the number of times the SMT solver is asked for solutions.

[5] The prototype is available for the PLDI reviewers in an anonymous repository at https://anonymous.4open.science/r/isla-pldi/. We will release it under an open source license after acceptance of this paper.

6 EVALUATION

To evaluate ISLa, we pose three central research questions:

RQ1 To which degree do ISLa constraints contribute to the *precision* of the input generator? With this question, we evaluate how much *benefit* does one get (in terms of more valid inputs) for how much *cost* (in terms of having to specify ISLa constraints).

RQ2 How *efficient* is the ISLa generator? Here, we evaluate whether using ISLa might be *prohibitive*—for instance, because it might be too slow for practical usage.

RQ3 How *diverse* are inputs generated from ISLa constraints? Here, we want to ensure that ISLa does not *overspecialize* (for instance, by producing only a small set of concrete inputs).

6.1 Evaluation Subjects

To evaluate ISLa, we identified frequently occurring context-sensitive language properties: (1) Declaration of identifiers (*def-use*), (2) redefinition—identifiers must not be declared more than once (*redef*), and (3) length or counting properties for binary inputs (*len-cnt*).

To cover these properties, we chose input languages of quite different character: (1) One highly structured (XML) and one more human-readable (reStructuredText (reST)) *markup language*, (2) a *data exchange format* (CSV), (3) a *programming language* (Scriptsize-C), and (4) a *binary format* (tar). The Scriptsize-C language is a proper subset of C which we derived from Tiny-C [7] by adding explicit variable declarations (in Tiny-C, all variables are globally declared by default). Scriptsize-C supports integer arithmetic expressions, comparisons, and **`if`**, **`while`** and **`do-while`** constructs. Excluded are, e.g., arrays, pointers, and function declarations.

For each of these languages, we defined *grammars* from their specification; for XML, we used a pre-existing grammar from the Fuzzing Book [29] which we refined with additional context-free constraints. We then added semantic constraints to all of those can in ISLa.

The tar archive format represents properties of binary inputs; which comes with strict length constraints (block sizes), and requires the computation of a checksum. Checksums are generally out of scope of SMT-LIB, which is why we spent 15 lines of Python code to implement a dedicated semantic predicate for tar checksums.

Finally, for ground truth, for each language, we chose a test target which processes them. Table 1 gives an overview of languages, test targets, and properties.

Table 1: Overview of evaluation targets and their properties. Properties in italic font are not covered by our specification.

Language	Test Target	*def-use*	*redef*	*len-cnt*	other
Scriptsize-C	`clang`	✓	✓	✗	*Nontermination Overflow*
XML	`xml.etree`	✓	✓	✗	Balance
tar	`tar`	✓	✓	✓	Checksum
reST	`rst2html`	✓	✓	✓	Numbering (∗)
CSV	`csvlint`	✗	✗	✓	✗

(∗) There are two further reST properties we did not consider, but which are in principle in the scope of ISLa: Alignment of continuing text in bullet lists, and same relative order of footnote references and footnotes for references without autonumber labels.

Table 2: ISLa Efficiency, Precision, and Input Diversity

	Constraints	LOC constraints (all)	Efficiency Inputs/s	Precision %valid	Diversity %k-paths
C	(none)	—	3.4	24	55
	+ no-redef	4 (11)	1.7	16	63
	+ def-use	6 (14)	2.3	100	74
XML	(none)	—	4.1	17	95
	+ namespace	12 (32)	2.9	47	95
	+ balance	2 (9)	7.9	100	98
	+ no-redef	5 (12)	0.4	100	94
TAR	(none)	—	6.1	0	N/A
	+ length	40 (132)	0.1	0	N/A
	+ checksum	3 (12)	0.4	26	N/A
	+ reference	14 (26)	0.5	100	N/A
reST	(none)	—	3.0	66	90
	+ reference	6 (15)	5.8	89	95
	+ length	7 (17)	10.2	98	77
	+ numbering	7 (15)	9.7	99	84
	+ no-redef	4 (10)	14.4	100	85
CSV	(none)	—	6.5	62	96
	+ columns	7 (14)	4.8	100	98

The "Efficiency" column considers *all* produced inputs; "Precision" and "Diversity" consider *valid* (accepted) inputs.

6.2 RQ1: Precision

The aim of ISLa is to produce more valid inputs, at the effort of specifying input constraints. Since ISLa is closed under conjunction, specifications can be added until a satisfying precision is reached. Table 2 relates the lines of ISLa code we added to account for a semantic property and the resulting overall precision. The row "(none)" stands for "no constraint" added; in the "Precision" column, we see that only 17% of generated XML inputs are valid without constraints (essentially those without ⟨*xml-open-tag*⟩ pairs); for tar, not a single input is valid. All values are obtained from a one-hour run of the input generator.

For every constraint added, we provide its length in lines of code, first for the constraint itself (the lines within **`constraint`** clause), and then the overall length. For Listing 1 (*balance* in Table 2), these lengths are 2 and 9, respectively.

We see that with each additional constraint, the precision increases; for XML, a total of 19 lines of constraints (within 53 lines of ISLa spec) is required to achieve 100% precision, with the last one going from a value slightly below 100% towards the totality of all inputs.

- ✏ Few ISLa constraints suffice to obtain a high number of valid inputs.
- ✏ ISLa constraints can ensure that *all* inputs are valid.

The most verbose property is the "length" property for TAR, where each field of the archive has to conform to strict length bounds. Yet, the constraint consists of a conjunction of simple constraints (most of them two lines only). If we *do not* specify the length and checksum constraints, we cannot produce even a single valid tar file.

6.3 RQ2: Efficiency

Our next question concerns the *time effort* required to produce these valid inputs. In Table 2, the column "Efficiency" shows the number of inputs produced per second by the ISLa generator. As would be expected, having to solve more constraints results in fewer inputs being generated, but the time taken easily allows to produce a sizable test corpus within a few minutes.

- ✏ For practical settings and input languages, ISLa produces valid inputs in reasonable time.

Interestingly, for reST, we obtain *more* inputs *with* ISLa constraints; this is because the underlying non-constrained grammar fuzzer produces very long inputs.

Finally, being able to quickly produce more inputs, as with TAR, is not useful if all of these inputs are invalid. This also defines the main usage scenario of ISLa: to produce inputs which either are very complex or which take a long time to be checked by the program under test.

6.4 RQ3: Diversity

A test generator should produce inputs exercising different language features, by which one can expect to reach different paths in the language processor [9]. Essentially, 100 % precision can be reached by always producing the same, small input. To validate that ISLa generates *diverse* and thus *interesting* inputs, we compute the *accumulated k-path coverage* [9] of the generated inputs, assessing how many paths in the grammar of length k are present in a derivation tree. The higher the k-path coverage, the higher the diversity.

The "Diversity" column in Table 2 shows the percentage of accumulated *3-paths* during a one-hour run per all 3-paths in the grammar. For example, generating Scriptsize-C programs from the grammar only achieves 55 % coverage, while we cover 74 % of all 3-paths when we add the "no-redef" and "def-use" constraints. In all cases, we only count valid inputs accepted by the program under test; as we do obtain TAR results only when all constraints are in place, we do not have a meaningful comparison here.

Overall, the inputs produced by ISLa have the same diversity, if not better, than inputs produced without constraints; the outlier here is reST, where the overly long inputs generated (see above) cover several reST features. Still, with a 3-path coverage of more than 50%, we can mitigate the threat that ISLa-generated inputs would be overly specialized.

✏ Inputs produced by ISLa cover the diversity of the underlying grammar.

6.5 Threads to Validity

We supported our claim that ISLa is a useful specification language by expressing context-sensitive properties of five subject input languages. Whether indeed ISLa is sufficiently expressive and its solver sufficiently precise depends on whether our choice of subjects is representative. There is a potential threat of *overfitting*, i.e., that we designed ISLa to exactly fit the test subjects. We mitigate this threat by choosing diverse languages, i.e., not only programming or markup languages, or binary formats, but a *mixture* of those. Furthermore, we identified and clustered context-sensitive properties of the test subjects, which, in our opinion, supports the claim that those are representative and can be transferred to different targets.

7 RELATED WORK

7.1 Attribute Grammars

ISLa provides a framework to specify *input requirements*, or preconditions, of a program. It targets the system level, where inputs are generally strings. Popular parser generators like ANTLR[6] and the pioneer yacc [15] promoted the use of CFGs for the description of complex, structured inputs. However, specifications designed for *parsing* inputs are rarely specific enough to also be used for *producing* inputs, which is the gap that ISLa fills.

Attribute grammars [16] go beyond context-free grammars by associating grammar symbols with synthesized and derived attributes. This allows checking semantic properties or pre-process inputs, e.g., by converting them into intermediate structures; if attributes are expressed in a general-purpose programming language, one can express arbitrarily complex semantic properties. The meta-compiler JastAdd [10], for instance, supports imperative specifications using Java code. The same holds for ANTLR (Java) and yacc (C). ISLa's mix of quantifiers, structural predicates, and SMT-LIB assertions allows expressing important input properties and can be used for *parsing* and *producing* inputs alike.

[6] https://www.antlr.org/

7.2 Grammar-Based Test Generation

Context-Free Grammars are well-suited for syntax-aware test input generation. CSmith [28] and LangFuzz [12], for example, use CFGs as a basis to randomly create syntactically valid C and JavaScript programs, respectively; LangFuzz can also *parse* and *recombine* program fragments. CSmith discovered hundreds of bugs in C compilers, LangFuzz thousands of bugs in JavaScript infrastructures. Grammarinator [11] produces inputs from ANTLR grammars, and unveiled bugs in a JavaScript interpreter.

ISLa can be located in between Grammarinator and CSmith: It can produce inputs from *different language models* like Grammarinator, but fulfills *semantic properties* like CSmith. Yet, the probability that Grammarinator will create one valid tar file from a CFG approaches zero, and CSmith can only generate—well—C files. CSmith also falls in the category of semantics-aware test generation, which we discuss next.

7.3 Test Generation with Semantic Properties

FormatFuzzer [5] is a fuzzer for binary formats. It is parameterized with *binary templates* as language models, which resemble C structs, but come with added code for satisfying semantic constraints, including complex expressions, control statements and functions. These constraints are strictly *local*, though, mainly supporting checksums and length fields for binary formats. Non-local constraints, such as *def-use* properties, would have to be programmatically implemented as a symbol table; support for solving complex constraints is not available either. ISLa's constraints, in contrast, are declarative, can apply to arbitrary elements in the derivation tree, and are easily solved using Z3.

Pan et al. [20] use Higher-Order Attribute Grammars [27] for fuzz testing, providing custom predicates for parse tree manipulation (e.g., length constraints and checksum computation) in a general-purpose programming language. Compared to ISLa, context-sensitive selection (e.g., using quantifiers and match expressions) is not supported; if predicates should be applied at a finer granularity, the underlying grammar has to be changed. The approach neither supports parsing nor generation from scratch.

Dewey et al. [4] propose to express grammars and constraints using the Prolog Constraint Logic Programming (CLP) framework for language-based test generation. The idea is that Prolog, as a logic programming language, is relatively declarative and thus shares the idea of declarative constraints with ISLa. However, CLP-based language fuzzers also cannot be used for parsing, in contrast to ISLa (which is based on grammars).

7.4 Property-Based Testing

In contrast to fuzz testing on the system level, where inputs are basically strings, Property-Based Testing (PBT) (pioneered by QuickCheck [2]) automatically produces *data structures of the host language* to test individual function against a user-defined property. This allows using features of the considered programming language, which is not available when working with unstructured system inputs.

ProSyT [3] translates declarative properties specified in the *PropEr* language to Prolog/CLP and produces test inputs for Erlang functions. Like ISLa, PropEr separates semantic constraints from the "syntactic" structure of Erlang algebraic data types. However, the language is limited to integer constraints.

Luck [17] is a Haskell-based specification language for checking and generating from constraints. It integrates test predicates and QuickCheck [2] generators with little syntactic overhead. As a functional language with pattern matching and recursion, it differs from ISLa, which resembles first-order logic (similarly to PropEr and specification languages used for program proving, such as JML [14] and ACSL [22]).

The Zest [19] system converts existing input generators into *parametric* generators. By manipulating the untyped parameters based on program feedback, Zest creates syntactically valid input mutants exercising interesting program paths. In contrast to ISLa, however, it cannot be used for property checking or making semantic constraints explicit.

The central difference between ISLa and all PBT approaches is that ISLa operates at the system level, producing *system* inputs rather than internal data structures; the concept of *parsing* and *mutating* existing data is not present in PBT.

8 CONCLUSION AND FUTURE WORK

We proposed ISLa, a declarative specification language for context-sensitive constraints of system inputs. In our framework, syntactic language constraints are specified using Context-Free Grammars (CFGs), which are great for parsing, but often too coarse for generating inputs. Context-sensitive refinements are expressed by ISLa constraints, using the vocabulary defined by the CFG. We formally defined ISLa's syntax, semantics, and input generator, and demonstrated that our system can be used to generate semantically inputs significantly faster than by generating from a CFG alone.

We project to work at the following applications of ISLa:

Fuzzer integration. We plan to examine whether we can leverage evolutionary greybox fuzzers like AFL to boost the generation speed for ISLa specifications. Our idea is to use ISLa to generate high-quality *seed inputs*, and to integrate the ISLa *constraint checker* into the fuzzer's fitness function. Note that constraint checking is always significantly faster than constraint solving.

Learning specifications. Our mid-term vision is to automatically *learn* ISLa properties from sample inputs, controlled experiments, and potentially lightweight program analysis. Hypothesized input invariants can be systematically refined by generating concrete inputs from them and investigating the behavior of the target program. One direction is to explore simple pattern-based techniques à la Daikon [6]. Furthermore, we will explore more advanced *constraint synthesis* techniques inspired by advances in the area of program synthesis.

The ISLa prototype is available at

https://anonymous.4open.science/r/isla-pldi/.

REFERENCES

[1] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2017. *The SMT-LIB Standard: Version 2.6.* Technical Report. Department of Computer Science, The University of Iowa. Available at www.SMT-LIB.org.

[2] 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 (ICFP)*, Martin Odersky and Philip Wadler (Eds.). ACM, 268–279. https://doi.org/10.1145/351240.351266

[3] Emanuele De Angelis, Fabio Fioravanti, Adrián Palacios, Alberto Pettorossi, and Maurizio Proietti. 2019. Property-Based Test Case Generators for Free. In *Tests and Proofs - 13th International Conference, TAP@FM 2019, Porto, Portugal, October 9-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11823)*, Dirk Beyer and Chantal Keller (Eds.). Springer, 186–206. https://doi.org/10.1007/978-3-030-31157-5_12

[4] Kyle Dewey, Jared Roesch, and Ben Hardekopf. 2014. Language Fuzzing Using Constraint Logic Programming. In *ACM/IEEE International Conference on Automated Software Engineering, ASE 2014*, Ivica Crnkovic, Marsha Chechik, and Paul Grünbacher (Eds.). ACM, 725–730. https://doi.org/10.1145/2642937.2642963

[5] Rafael Dutra, Rahul Gopinath, and Andreas Zeller. 2021. FormatFuzzer: Effective Fuzzing of Binary File Formats. *CoRR* abs/2109.11277 (2021). arXiv:2109.11277 https://arxiv.org/abs/2109.11277

[6] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. 1999. Dynamically Discovering Likely Program Invariants to Support Program Evolution. In *Proceedings of the 1999 International Conference on Software Engineering, (ICSE)*, Barry W. Boehm, David Garlan, and Jeff Kramer (Eds.). ACM, 213–224. https://doi.org/10.1145/302405.302467

[7] Marc Feeley. 2001. Tiny-C Compiler. https://www.iro.umontreal.ca/~felipe/IFT2030-Automne2002/Complements/tinyc.c. Accessed: 2021-10-06.

[8] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-Based Whitebox Fuzzing. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI)*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 206–215. https://doi.org/10.1145/1375581.1375607

[9] Nikolas Havrikov and Andreas Zeller. 2019. Systematically Covering Input Structure. In *34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 189–199. https://doi.org/10.1109/ASE.2019.00027

[10] Görel Hedin and Eva Magnusson. 2001. JastAdd—A Java-Based System for Implementing Front Ends. *Electron. Notes Theor. Comput. Sci.* 44, 2 (2001), 59–78. https://doi.org/10.1016/S1571-0661(04)80920-4

[11] Renáta Hodován, Ákos Kiss, and Tibor Gyimóthy. 2018. Grammarinator: A Grammar-Based Open Source Fuzzer. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, Wishnu Prasetya, Tanja E. J. Vos, and Sinem Getir (Eds.). ACM, 45–48. https://doi.org/10.1145/3278186.3278193

[12] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proceedings of the 21th USENIX Security Symposium*, Tadayoshi Kohno (Ed.). USENIX Association, 445–458. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/holler

[13] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2007. *Introduction to Automata Theory, Languages, and Computation, 3rd Edition*. Addison-Wesley.

[14] Marieke Huisman, Wolfgang Ahrendt, Daniel Grahl, and Martin Hentschel. 2016. Formal Specification with the Java Modeling Language. In *Deductive Software Verification - The KeY Book - From Theory to Practice*, Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich (Eds.). Lecture Notes in Computer Science, Vol. 10001. Springer, 193–241. https://doi.org/10.1007/978-3-319-49812-6_7

[15] Stephen C Johnson. 1979. Yacc: Yet Another Compiler-Compiler. https://www.cs.utexas.edu/users/novak/yaccpaper.htm. Accessed: 2021-11-19.

[16] Donald E. Knuth. 1990. The Genesis of Attribute Grammars. In *Proceedings of the International Conference on Attribute Grammars and their Applications (Lecture Notes in Computer Science, Vol. 461)*, Pierre Deransart and Martin Jourdan (Eds.). Springer, 1–12. https://doi.org/10.1007/3-540-53101-7_1

[17] Leonidas Lampropoulos, Diane Gallois-Wong, Catalin Hritcu, John Hughes, Benjamin C. Pierce, and Li-yao Xia. 2017. Beginner's Luck: A Language for Property-Based Generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL) 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 114–129. https://doi.org/10.1145/3009837.3009868

[18] Barton P. Miller, Lars Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (1990), 32–44. https://doi.org/10.1145/96267.96279

[19] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic Fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, Dongmei Zhang and Anders Møller (Eds.). ACM, 329–340. https://doi.org/10.1145/3293882.3330576

[20] Fan Pan, Ying Hou, Zheng Hong, Lifa Wu, and Haiguang Lai. 2013. Efficient Model-based Fuzz Testing Using Higher-order Attribute Grammars. *J. Softw.* 8, 3 (2013), 645–651. https://doi.org/10.4304/jsw.8.3.645-651

[21] Alan J. Perlis. 1982. Epigrams on Programming. *ACM SIGPLAN Notices* 17, 9 (1982), 7–13. https://doi.org/10.1145/947955.1083808

[22] The Frama-C Authors. 2021. ANSI/ISO C Specification Language. https://www.frama-c.com/html/acsl.html. Accessed: 2021-11-13.

[23] The OSS-Fuzz Contributors. 2021. OSS-Fuzz: Continuous Fuzzing for Open Source Software. https://github.com/google/oss-fuzz/blob/master/README.md. Accessed: 2021-10-06.

[24] The SMT-LIB Initiative. 2021. SMT-LIB Theories. http://smtlib.cs.uiowa.edu/theories.shtml. Accessed: 2021-10-19.

[25] Cesare Tinelli, Clark Barrett, and Pascal Fontaine. 2020. Theory Strings (SMT-LIB Version 2.6). http://smtlib.cs.uiowa.edu/theories-UnicodeStrings.shtml. Accessed: 2021-10-07.

[26] Dirk van Dalen. 1994. *Logic and Structure (3rd ed.)*. Springer.

[27] Harald Vogt, S. Doaitse Swierstra, and Matthijs F. Kuiper. 1989. Higher-Order Attribute Grammars. In *Proceedings of the ACM SIGPLAN'89 Conference on Programming Language Design and Implementation (PLDI)*, Richard L. Wexelblat (Ed.). ACM, 131–145. https://doi.org/10.1145/73141.74830

[28] 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 (PLDI) 2011*, Mary W. Hall and David A. Padua (Eds.). ACM, 283–294. https://doi.org/10.1145/1993498.1993532

[29] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2021. Grammar Coverage. In *The Fuzzing Book*. CISPA Helmholtz Center for Information Security. Accessed: 2021-11-13.

APPENDIX

2 ISLa by Example

In Section 2, we discussed the semantic properties *def-use* and *redefinition* along the XML language. Apart from those, there are two other re-occurring *generic* constraints we would like to discuss: *Length* properties and complex conditions for which we need dedicated *semantic predicates*.

One of the target languages in our evaluation (Section 6) is reStructuredText (reST), a plaintext markup language used, e.g., by Python's docutils or the documentation generator Sphinx[7]. In reST, document (sub)titles are underlined with "=" or "-" symbols. However, titles are only valid if the length of the underline is not smaller than the length of the title text. This property cannot be expressed in a CFG; however, we can easily capture it in an ISLa constraint:

```
constraint {
  forall title="{titletxt}\n{underline}" in start:
    (>= (str.len underline) (str.len titletxt))
}
```

There are properties which cannot be expressed using structural predicates and SMT-LIB formulas alone. A stereotypical case are checksums occurring in many binary formats, such as in the tar archive file format from our benchmark set. To account for such situations, we can extend the ISLa language with additional atomic assertions, so-called *semantic predicates*. In contrast to structural predicates such as `inside` or `same_position`, which we have seen before, semantic predicates do not always evaluate to false for invalid arguments. Instead, they can suggest a *satisfying solution*. The solver logic for individual semantic predicates is implemented in Python code in our prototype. Once this logic has been implemented, we can pass such predicates as additional signature elements to both the ISLa evaluator or solver and use them in constraints. The following constraint, which is part of our constraint set for tar files, uses a semantic predicate `tar_checksum` computing a correct checksum value for the header of a tar file.

```
constraint {
  forall header in start:
    forall checksum in header:
      tar_checksum(header, checksum)
}
```

Another use case for semantic predicates is when the SMT solver frequently times out when looking for satisfying assignments. This happens in particular for constraints involving a complex combination of arithmetic and string (e.g., regular expression) constraints. For example, valid CSV files have the property that all rows have the same numbers of columns. Assuming that we know the number of columns in the file header, we could create a regular expression matching all CSV lines with the same number of columns. However, if we admit quoted expressions and a wide character range for contained text, these regular expressions get quite complex, and the problem exceeds the capabilities of current SMT solvers in our experience. Thus, we implemented a new semantic predicate `count` which counts the number of oc-

[7] https://www.sphinx-doc.org/

Listing 3: Greedily matching match expressions.

```
1  def match(tree, mexp, result, excluded=()):
2      subtrees = tree.subtrees(PREORDER)
3      curr_var = pop(mexp)
4
5      while subtrees and curr_var:
6          path, subtree = pop(subtrees)
7          if (subtree.value != curr_var.n_type
8              or (path, curr_var) in excluded):
9              continue
10
11         result[curr_var] = (path, subtree)
12         curr_var = pop(mexp, None)
13
14         subtrees = [
15             (p, s) for p, s in subtrees
16             if not p[:len(path)] == path]
17
18     return not subtrees and not curr_elem
```

we discuss later). Before calling `match`, terminal and nonterminal symbols in the match expression are converted to "dummy" variables of a randomly chosen, fresh name. For terminal symbols, the type of these variables (which can be accessed via the field "`n_type`") is the symbol itself. Furthermore, `match` is only called after "flattening" the match expression. That is, we compute all combinations of activated and non-activated *optional* expressions. If there are n optionals in the match expression, we obtain 2^n flattened lists consisting of variables only. Thus, the match expression `mexp` which `match` receives is a list of (ordinary and dummy) variables, while original match expressions consist of bound variables, terminal and nonterminal symbols, and, if there are *optional* elements, non-nested lists of terminal and nonterminal symbols. We enumerate all paths and corresponding subtrees in the tree in pre-order (Line 2). As long as there are still subtrees and bound elements to match (Line 5), we take the next subtree (Line 6) and check whether that tree matches the current tree label (Line 7). If this is the case (we ignore the check in Line 8 for now), we record the match for the current bound element (Line 11) and remove all paths below that subtree (Lines 14 to 16). The pointer for the current element of the match expression (`curr_var`) is set to the next bound variable or **`None`** (passed as an optional default value to the `pop` function) if there are no more elements to match. A match is *complete* (and **`True`** is returned in Line 18) if there do not remain unmatched subtrees and the current element pointer is **`None`**; otherwise, we return **`False`**.

This greedy match procedure sometimes fails for structures with recursive nonterminals. Consider the case of a compound XML attribute `<attribute> <attribute>`. If we try to match this expression against a derivation tree whose root is also `<attribute>`, `match` unifies the *root* with the first `<attribute>` occurrence in the match expression, leading to an unsuccessful match. Thus, `match` is called by an outer backtracking procedure which excludes selected matches from an incomplete `result` assignment, which are then avoided in Line 8 in subsequent calls of `match`.

In the following, we use `match` as *match*(t, *mexpr*) for a

solutions as assumptions), we assume that we get a *set* of assignments of tree identifiers to new subtrees from σ.

Semantic Predicates. We divide the set PSym of predicate symbols into two disjoint sets PSym_{st} and PSym_{sem} of *structural* and *semantic* predicates. Structural predicates address structural constraints, such as *before* or *within*. They evaluate to $\top$ or $\bot$. *Semantic* predicates formalize more complex constraints, such as specific checksum implementations. In addition to $\top$ or $\bot$, semantic predicate formulas may evaluate to a set of assignments, as in the case of satisfiable SMT expressions, or to the special value "not ready" (denoted by $\circlearrowleft$). Intuitively, an evaluation results in $\top$ ($\bot$) if all of (not any of) the concrete derivation trees represented by an abstract tree satisfy the predicate. A set of assignments is returned if there are reasonable "fixes" of the tree (e.g., all elements relevant for a checksum computation are determined, such that the checksum can be computed by the predicate). One may obtain $\circlearrowleft$ if the constrained tree lacks sufficient information for such a computation; for instance, we cannot compute a checksum if the summarized fields are still abstract.

In contrast to all other constraint types, the *order of semantic predicate formulas within a conjunction matters* (we use ordered sets in the implementation of our CDTs). The reason is that each semantic predicate comes with its own, atomic solver. Consider, for example, a binary format which requires a semantic predicate for the computation of a data field (e.g., requiring a specific compression algorithm) and another one for a checksum which also includes the data field. Then, one must *first* compute the value of the data field, and *then* the value of the checksum. Changing this order would result in an invalid checksum. Since SMT formulas are composable, we recommend using semantic predicates only if the necessary computation can either not be expressed in SMT-LIB, or the solver frequently times out when searching for solutions.

Tree Insertion. Existential constraints can occasionally be solved by matching them against the indicated subtree, similarly to universal quantifiers. In general, though, we have to manipulate the tree to enforce the existence of the formalized structure. If a successful match is not possible, we therefore constructively *insert* a new tree into the existing one. The function *makeTree*(v) creates a new derivation tree consisting of a single root node of type *vtype*(v). When passing it a match expression *mexpr* as additional argument, it creates a minimal open tree rooted in a node of type *label*(v) and matching *mexpr*. The function *insert*(t', t) tries to inserts the tree t' into t. Whether this is possible entirely depends on t, t' and the grammar. In the simplest case, t has an open leaf from which the nonterminal *label*(t') is reachable. Then, we create a suitable tree connecting the leaf and the root of t' and glue these components together.

If this is not possible, we attempt to exploit recursive definitions in the grammar. Consider, for example, a partial XML document according to the grammar in Fig. 1 and the constraint **`exists`** `optag` **`in`** `tree: (= optag "<a>")`, where `optag` is of type ⟨*xml-open-tag*⟩ and `tree` points to a node with root of type ⟨*xml-tree*⟩. If there is some opening tag of form `<a>` in *tree*, we can eliminate the constraint.

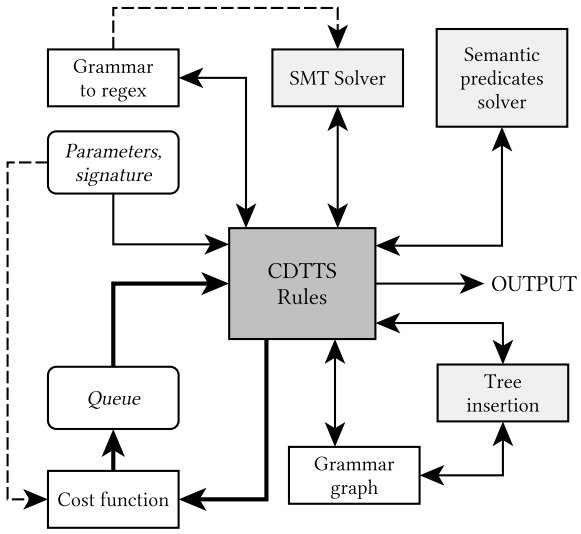

Figure 4: Schematic representation of the ISLa solver. The main component is the CDTTS implementation, which receives as inputs (nodes in italic font) a signature, configuration parameters, and an initial queue element. The main solver loop is highlighted with bold arrow lines. Rectangles with light gray background are main constraint solving components; the remaining ones are auxiliary components.

tree insertion itself) are trivially to prove, because we add the additional constraint again to the constraint set.

Finally, Rules (15) and (16) consider more concrete trees and are therefore precise for the same reasons as Rule (8). □

Theorem 4.8. *The ISLa CDTTS in Fig. 3 is globally complete.*

Proof Sketch. To prove the global completeness of our system, we have to show that the semantics of each input CDT is contained in the semantics of all reachable CDTs with empty constraint set. We reduce this problem as follows. First, we show *local* completeness, i.e., that no information is lost by applying any transition rule of our CDTTS. Second, we argue that for each *valid* CDT, there is an applicable rule in the CDTTS. Third, we argue that for each input CDT, there is *one* output CDT which is *closer* to a state with empty constraint set in the CDTTS than the inputs. From this, we conclude global completeness as follows: Since for each valid state, there is a transition step from which get closer to an output state with empty constraint set, this also holds for each valid state produced by this step. By additionally requiring that the individual steps do not lose information, we conclude global completeness.

We argue for the local completeness of a chosen set of CDTTS rules.

The expansion and finishing rules are locally complete if *all* expansions are considered. This is the case in our CDTTS,

Our cost function computes the weighted geometric mean of *cost factors* cf_i and corresponding *weights* w_i as

$$cost = \left(\prod_{i=1,\ldots,n}^{w_i \neq 0} (cf_i + 1)^{w_i} \right)^{\left(\sum_{i=1,\ldots,n}^{w_i \neq 0} w_i\right)^{-1}} - 1$$

We filter out pairs of cost factors and weights where the weight is 0; in this case, the corresponding cost factor is deactivated. Furthermore, we avoid the case that the final cost value is 0 if one of the factors is 0 by incrementing each factor by 1, and finally decrementing the result by 1 again.

We chose the following cost factors:

Tree closing cost. We precompute, for each nonterminal in the grammar, an approximation of the instantiation effort of that nonterminal, roughly by instantiating it several times randomly with a fuzzer, and then summing up the sizes of the possible grammar expansion alternatives in the resulting tree. The closing cost for a derivation tree is defined as the sum of the costs of each nonterminal symbol in all open leaves of the tree.

Constraint cost. Certain constraints are more expensive to solve than others. In particular, solving existential quantifiers by tree insertion is computationally costly. This cost factor assigns higher cost for constraints with existential and deeply nested quantifiers.

Derivation depth penalty. As the solver's queue fills up, it becomes more improbable for individual queue elements to be selected next. If we assign a cost to the derivation depth, it becomes more likely that the solver eventually comes back to partial solutions discovered earlier, avoiding starvation of such inputs.

k-path coverage. When choosing between different partial trees, we generally want to generate those exercising more language features at once. The k-path coverage metric [9] computes all paths of length k in a grammar and derivation tree; the proportion of such paths covered by a tree is then the coverage value. We penalize trees which cover only few k-paths. The concrete value of k is configurable; the default is 3.

Global k-path coverage. For each final result produced by the solver, we record the covered k-paths and from then on prefer solutions covering *additional* language features. Once all k-paths in a grammar have been covered, we erase the record.

The influence of these cost factors can be controlled by passing a tuple of weights to the solver. We provide a reasonable default vector $((11, 3, 5, 20, 10))$, but in certain cases, a problem-specific tuning might be necessary to improve the performance. Our implementation provides an evolutionary parameter tuning mechanism, which runs the solver with randomly chosen weights, and then computes several generations of weight vectors using crossover and mutation. The fitness value of a weight vector is determined by the generation speed, a vacuity estimator, and a k-path-based coverage measure.