Input Invariants

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 variable has to be defined before it is used," "the length of the 'file name' block in a TAR file has to equal 100 bytes," or "the number of columns in all CSV 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 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.*

ISLa constraints can be specified manually, but also *mined* from existing input samples. For this, our ISLearn prototype uses a catalog of common patterns (such as the ones above), instantiates these over input elements, and retains those candidates that hold for the inputs observed and whose instantiations are fully accepted by input-processing programs. The resulting constraints can then again be used for fuzzing and parsing.

In our evaluation, we show that a few ISLa constraints suffice to produce inputs that are 100% semantically valid while still maintaining input diversity. Furthermore, we confirm that ISLearn mines useful constraints about definition-use relationships and (implications between) the existence of "magic constants", e.g., for programming languages and network packets.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; *Specification languages*; **Constraint and logic languages**; *Syntax*; *Semantics*; **Parsers**; *Software reverse engineering*; *Documentation*; • **Theory of computation → Grammars and context-free languages**; *Formalisms.*

KEYWORDS

fuzzing, specification language, grammars, constraint mining

1 INTRODUCTION

Automated software testing with random inputs (*fuzzing*) [19] is an effective technique for finding bugs in programs. Pure random inputs can quickly discover errors in input processing. Yet, 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.

Language-based fuzzers [8, 12, 13] 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 (`<O L="cmV">` ↪ `B7</O>`) contained a matching tag pair. This result is typical for language-based fuzzers used with a language specification designed for *parsing* 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 *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* inputs, which is a common feature of modern fuzzers. *Unrestricted grammars* can in principle specify any computable input property, but we see them as "Turing tar-pits," in which "everything is possible, but nothing of interest is easy" [22]—just try, for instance, to express that some number is the sum of two input elements. Finally, one could also replace CFGs by a *different formalism;* but this would mean to renounce a concept that many developers know (e.g., from the ANTLR parser generator or RFCs).

In this paper, we bring forward a different solution by proposing a (programming and target) *language-independent*, *declarative* specification language named *ISLa* (**I**nput **S**pecification **La**nguage) 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

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

⟨*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)

Listing 1: ISLa constraint for well-balanced XML expressions.

```
forall <xml-tree> tree="<{<id> opid}[<xml-attribute>]><inner-xml-
    ↪ tree></{<id> clid}>" in start:
  (= opid clid)
```

until the represented language is a sufficiently close approximation of the target language—an *invariant* over all inputs.

To get an idea of ISLa constraints, consider Listing 1, referring to the grammar in Figure 1. The constraint expresses that for all XML trees composed of an opening tag with ⟨*id*⟩ `opid` and a closing tag with ⟨*id*⟩ `clid`, both ⟨*id*⟩s have to be equal, as expressed by the SMT formula "`(= opid clid)`". This is typical for an ISLa constraint: It first *identifies* individual elements in the derivation tree, and then poses *constraints* over these elements. During fuzzing, ISLa would then produce matching pairs of opening and closing ⟨*id*⟩s.

The resulting valid inputs can be used as *seed inputs* for mutational fuzzers like AFL. ISLa's constraint checker can be integrated into the *fitness function* of evolutionary fuzzers, guiding their mutations toward semantically valid inputs; ISLa's checker can quickly reject invalid inputs without having to run actual tests.

While a *grammar* for ISLa can be extracted from inputs [17] and programs [9], where would ISLa *constraints* come from? Testers can write ISLa constraints manually, thus ensuring input validity, and add additional constraints to further control the inputs generated. However, they can also *mine constraints from existing inputs.* To this end, our *ISLearn* tool uses a catalog of common *constraint patterns*, instantiates these over all inputs and input elements, and retains those constraint candidates that hold for all inputs. The catalogue holds patterns to identify matching elements, length relations, arithmetic relations, checksums, and more. ISLearn is similar in spirit to the *Daikon* function-level invariant detector [6]. On top, ISLearn can automatically *verify* and *refine* constraint candidates by having the program under test check whether derived concrete inputs are valid. ISLearn can thus infer precise XML constraints from existing XML inputs, including the one in Listing 1.

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). To the best of our knowledge, ISLa is the first formalism for expressing context-sensitive constraints of system inputs.

A precise input generator. We describe an efficient procedure to generate inputs satisfying ISLa constraints (and their grammars), and discuss our implementation (Section 4). To the best of our knowledge, ISLa is the first fuzzer (and checker) to make use of such constraints, giving users unprecedented means to specify which system inputs should be generated.

Mining input constraints. We introduce ISLearn, a system for automatically mining input constraints in disjunctive normal form based on a configurable pattern catalogue (Section 5). To the best of our knowledge, ISLearn is the first approach to infer such invariants from given system inputs.

ISLa and its constraints are effective. In our evaluation (Section 6), we formalize semantic properties from diverse languages, 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 generate *100% precise* inputs while maintaining *diversity.* On top, our constraint miner ISLearn can extract precise invariants about ICMP packets, DOT graphs, and Racket programs.

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

2 ISLA BY EXAMPLE

Let us illustrate the expressive power of ISLa by detailing our XML example. When randomly passing inputs generated from the XML grammar in Fig. 1 using a grammar fuzzer 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 can *incrementally* refine the specification simply by adding individual input constraints 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, and about 20% because of a mismatched tag. Let us address these.

2.1 Matching Tags

The ISLa constraint in Listing 1 addresses the problem of *mismatched tags* by enforcing that the two IDs match. It uses a universal quantifier (**`forall`**), quantifying over all sub expressions of type ⟨*xml-tree*⟩, which is the specified type of the bound variable `tree`. Types are nonterminals from the reference grammar (here the XML grammar in Fig. 1) or the special type `int` for quantifiers over numbers. The present quantifier uses *pattern matching.* ISLa only considers matches conforming to the pattern (in quotation marks); in the case of a successful match, not only the quantified variable `tree`, but also the variables in the pattern (in curly braces) are *bound* to the corresponding parts of the matched input segment. Match expressions may contain optional elements in square brackets to capture multiple expansion alternatives. The core of the **forall** formula is an *SMT-LIB S-expression* stating that variables opid and clid are equal. Since ISLa extends the SMT-LIB language [24], it supports all its string constraints. An ISLa constraint contains exactly one constant symbol, which determines the type of the inputs described by the constraint. By default, this is a symbol start of type ⟨*start*⟩, which can be customized by a declaration **const** name: type; before the actual constraint.

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.

2.2 Binding Prefixes

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 the same or an outer 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" 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.

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.

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

```
1 forall <xml-attribute> attribute in start:
2   forall <id-with-prefix> prefix_id=
3       "{<id-no-prefix> prefix_use}:<id-no-prefix>" in attribute:
4     ((= prefix_use "xmlns") or
5       exists <xml-tree> outer_tag="<<id> {<xml-attribute> cont_attr}>
          ↪ <inner-xml-tree></<id>>" in start:
6         (inside(attr, outer_tag) and
7          exists <xml-attribute> def_attr="xmlns:{<id-no-prefix>
              ↪ prefix_def}=\"<text>\"" in cont_attr:
8            (= prefix_use prefix_def)))
```

The ISLa code quite literally follows the natural language specification we described previously, except that we specialized it to only quantify over *attributes* (Line 1) and generally permit the xmlns prefix (Line 4) using a *disjunction*: Either the prefix is xmlns, *or* it must be explicitly defined.

2.3 Targeted Testing

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:

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

```
forall <xml-attribute> attribute="web:<id-no-prefix>={<text> 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 developers or security researchers, as a regression test validating a fix for a potential exploit:

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

2.4 Mining Constraints

Constraints like the ones described above can also be *mined* from existing inputs. For this purpose, we create a *schematic* version of the constraint in Listing 1 which is *independent from the choice of a particular grammar:*

```
forall <?NONTERMINAL> container="{<?MATCHEXPR(opid, clid)>}"
  in start: (= opid clid)
```

This pattern can be added to the catalog of our ISLearn system, enabling the system to infer similar constraints for a different grammar. The placeholder <?NONTERMINAL> represents any nonterminal in that grammar; <?MATCHEXPR(opid, clid)> represents any suitable match expression for an instantiation of container, containing two nonterminal occurrences that are bound to variables opid and clid. ISLearn generates candidate instantiations from such patterns, and then filters those that hold for a set of given or automatically generated sample inputs. Hence, given a set of XML inputs, ISLearn can easily learn the constraint in Listing 1.

To avoid overspecialization toward a small set of inputs, ISLearn can automatically *validate* constraint candidates—by generating further inputs from them and checking whether these inputs would be accepted by the program. This also works in *debugging* scenarios: If we have a set of (XML) inputs for which a specific property holds (say, the length of some input element exceeds some constant), ISLearn will not only learn that constraint, but can also ensure that further instantiations of the constraint reproduce the failure.

2.5 Summary

With these examples, we have demonstrated how ISLa constraints precisely characterize input classes associated to some program behavior. Developers can make use of these descriptions to obtain *semantically valid* inputs, to *describe* the conditions of discovered bugs, for targeted *triggering* of such bugs. Given existing inputs, ISLearn can determine constraints that precisely characterize input properties and program behavior.

Note that *without* ISLa and ISLearn, implementing any of these constraints 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* 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

ISLa constraints are built from a *signature* of grammar, predicate, and variable symbols. We first formally define CFGs, following [14, 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 `int` for *numeric* variables representing 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 $n \in N \cup \{\texttt{int}\}$, $\texttt{int} \notin N$.

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 ($\texttt{f}\ a_1\ \ldots\ a_n$), 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 [23].

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) (**not** φ), (φ **and** ψ), (φ **or** ψ) are in Fml for $\varphi, \psi \in \text{Fml}$.
(4) **forall** *type* x **in** y: φ and **exists** *type* x **in** y: φ are in Fml for $x, y \in \text{VSym}$, $vtype(x) = type \in N \cup \texttt{int}$, and $\varphi \in \text{Fml}$.
(5) **forall** *type* x="*mexp*" **in** y: φ and its existential counterpart **exists** *type* x="*mexp*" **in** y: φ are in Fml, where $x, y \in \text{VSym}$, $vtype(x) = type \in N$, and $\varphi \in \text{Fml}$, and *mexp* is a string consisting of symbols in $N \cup T$, non-nested lists of such symbols (*optional* symbols), and variables $v \in \text{VSym}$.

We use φ **implies** ψ as a shorthand for (**not** φ) **or** ψ.

The set Fml is relative to a signature Σ, left implicit for simplicity. Parentheses can be omitted according to the following precedence rules: Quantifiers bind stronger than negation, which binds stronger than conjunction, which binds stronger than disjunction.

We only consider (top-level) ISLa formulas which contain *exactly one* unbound variable, which is the default `start` variable, or the one specified in the optional **const** declaration.

Semantics. The semantics of an ISLa constraint are all strings derivable from the reference grammar that satisfy the constraint. To make this precise, we first define derivation trees. Then, we fix the meaning of ISLa constraints by defining a validation judgment.

Definition 3.4 (Derivation Tree). A *derivation tree* for a CFG $G = (N, T, P, S)$ is a *rooted ordered tree* such that (1) all vertices v are *labeled* with symbols $label(v) \in N \cup T$, where the root is labeled with S, (2) if $v_1, \ldots, v_k$ are the children of a node labeled with n, then there is a production $(n, s_1, \ldots, s_k) \in P$ such that for all v_i, $label(v_i) = s_i$. For a derivation tree t, we write $leaves(t)$ for the set of its leaves, 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. We define $closed(t) := \forall l \in leaves(t) : l \in T$, and $open(t) := \neg closed(t)$. $\mathcal{T}(G)$ is the set of all (closed and open) derivation trees for G.

Example 3.5. Fig. 2 visualizes the derivation tree of the XML document `<a>x</a>` for the XML grammar in Fig. 1: The root of the tree, v_1, is labeled with the grammar's start symbol $\langle xml\text{-}tree \rangle$; its edges 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 Definition 3.4, there has to be a production ($\langle xml\text{-}open\text{-}tag \rangle$, '<', $\langle id \rangle$, '>') 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 leaves $leaves(t)$ are $\{v_6, v_{13}, v_8, v_9, v_{10}, v_{14}, v_{12}\}$. 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).

We convert a derivation tree to a string by concatenating its leaves in order of their occurrence. We write $str(t)$ for the string obtained from t. If t is the tree from Fig. 2, we have $str(t) =$ `<a>x</a>`.

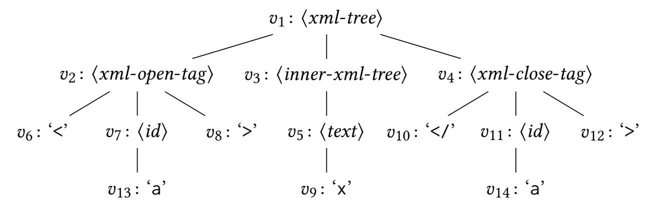

Figure 2: Example XML derivation tree.

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 $match(t, mexpr)$, where t is a tree and $mexpr$ a match expression. It returns a 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. For optional elements in match expressions, it considers all possibilities of present and non-present optionals.

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 $\{v \mapsto t\}$. By $\beta^{\downarrow}$ we denote 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, $\top$ and $\bot$ represent semantic truth and falsity, resp. Note that we expect σ to always return $\top$ or $\bot$. Timeouts, not uncommon for SMT solvers, are usually no problem for concrete formulas without free variables. Should the solver time out anyway, we interpret this as $\bot$.

Definition 3.6 (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** *type 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 $type \in N$.
(7) $\pi, \sigma, \beta \models$ **forall int** n **in** φ: iff $\pi, \sigma, \beta[n \mapsto t] \models \varphi$ holds for all trees t such that $str(t)$ represents a number in $\{0, 1, 2, \ldots\}$.
(8) $\pi, \sigma, \beta \models$ **exists** *type 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 $type \in N$.
(9) $\pi, \sigma, \beta \models$ **exists int** n **in** φ: iff $\pi, \sigma, \beta[n \mapsto t] \models \varphi$ holds for some tree t such that $str(t)$ represents a number in $\{0, 1, 2, \ldots\}$.
(10) $\pi, \sigma, \beta \models$ **forall** *type v*="*mexpr*" **in** w: φ iff $\pi, \sigma, \beta[v \mapsto t][m] \models \varphi$ holds for all subtrees t with root r in $\beta(w)$ such that $label(r) = type$ and there is a match $m = match(t, mexpr)$.
(11) $\pi, \sigma, \beta \models$ **exists** *type v*="*mexpr*" **in** w: φ iff $\pi, \sigma, \beta[v \mapsto t][m] \models \varphi$ holds for a subtree t with root r in $\beta(w)$ such that $label(r) = type$ and there is a match $m = match(t, mexpr)$.

Example 3.7. 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 (10) of Definition 3.6 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 the case.

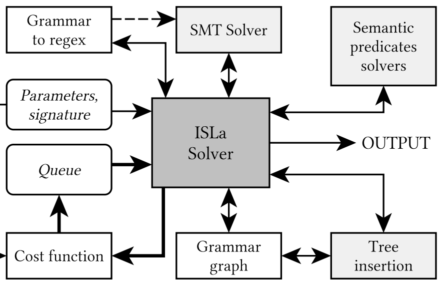

Figure 3: ISLa solver schema. Bold arrow lines depict the main solver loop. Light gray rectangles are main constraint solving components; the remaining ones are auxiliary components.

Definition 3.8 (ISLa Semantics). Let $\varphi \in \text{Fml}$ be an ISLa formula with the single free variable c for the signature $(G, \text{PSym}, \text{VSym})$, and π, σ be interpretations for predicates and SMT formulas. We define the semantics $[\![\varphi]\!]$ of φ as

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

4 SOLVING ISLA CONSTRAINTS

Our ISLa solver stepwise expands elements from a queue of *Conditioned Derivation Trees (CDTs)*. A CDT is a pair $\Phi \triangleright t$, where Φ is a set of ISLa formulas and t a—possibly open—derivation tree. Intuitively, the conjunction of the formulas in Φ constrains the inputs represented by t, similarly as $[\![\varphi]\!]$ constrains the language of the grammar. Open trees represent the possibly infinite set of derivation trees that can be derived from them by expansion according to the grammar rules; imposing constraints potentially reduces the set of applicable rules and thus the represented concrete trees. On the other hand, *closed* derivation trees only stand for themselves. If a constraint is added to a closed tree, the result is either empty (if the tree does not satisfy the constraint) or consists of the tree itself.

To facilitate references to trees in constraints, we assign unique, numeric identifiers to derivation tree nodes. These identifiers may be used instead of *free* variables in ISLa formulas (variables bound by quantifiers may not be replaced with tree identifiers).

Consider, for example, the ISLa constraint

$$\varphi = \textbf{forall}\ \langle id \rangle\ id\ \textbf{in}\ start\text{:}\ \texttt{(= (str.len}\ id\texttt{) 3)}$$

constraining the XML grammar in Fig. 1 to identifiers of length 3. The variable *start* occurs free in that formula. Let t be a tree consisting of a single (root) node with identifier 1, and labeled with $\langle start \rangle$. Then, $[\![\varphi]\!]$ is identical to the strings represented by the CDT

$$\{\textbf{forall}\ \langle id \rangle\ id\ \textbf{in}\ 1\text{:}\ \texttt{(= (str.len}\ id\texttt{) 3)}\} \triangleright t.$$

Fig. 3 schematically represents the ISLa constraint solver. Starting with the CDT above, the solver expands the open tree *t* according to the grammar and adds the resulting CDT into the queue. The queue itself is a *priority queue.* The order of CDTs inside the queue is determined by a configurable *cost function.*

Expansion continues as long as it gets us nearer to matching the universal quantifier (in the example, until another ⟨*id*⟩ nonterminal symbol is contained in the represented trees of the expansion). Eventually, the following state will be added to the queue:

{**forall** ⟨*id*⟩ *id* **in** *1*: `(= (str.len` *id*`) 3)`} ▹ `<`⟨*id*⟩`/>`

Now, the universal quantifier matches and is instantiated. Let 4 be the identifier of the subtree labeled with the ⟨*id*⟩ nonterminal. Then, we obtain (using bold font for the tree identifier in the SMT formula resulting from the instantiation):

{`(= (str.len` **4**`) 3)`,
forall ⟨*id*⟩ *id* **in** *1*: `(= (str.len` *id*`) 3)`} ▹ `<`⟨*id*⟩`/>`

The solver now removes the quantified formula from the constraint set, since there is no chance of obtaining *another* ⟨*id*⟩ by further instantiation. Next, it invokes the SMT solver to obtain a solution for the formula `(= (str.len` **4**`) 3)`. If we would simply ask the solver for a string of length 3, we would not necessarily receive an answer matching the language of the ⟨*id*⟩ nonterminal; for example, the solver could produce a sequence of three space characters. Thus, we use the "Grammar to regex" component to produce a regular expression describing the desired syntax, which we add to the solver query. While generally, it is not possible to *precisely* transform a CFG into a regular expression, it is often feasible for small sub grammars, like the one for the ⟨*id*⟩ nonterminal. Otherwise, we create an approximate regular expression by unfolding problematic recursions up to a fixed bound.

The solution returned by the solver is parsed into a derivation tree and substituted for the subtree with identifier 4; the SMT formula is removed from the constraint set. This results in a set of CDTs (the number of solutions requested from the SMT solver is configurable) with empty constraint sets and closed trees such as {} ▹ `<abc/>`. Since there are no constraints and open tree leaves left, `<abc/>` is immediately output as a solution of the constraint.

The solver not only stops tree expansion if it can be sure that no universal quantifier can eventually be matched by doing so; it also only expands open subtrees for which this is the case. Consequently, there are situations where the constraint set is empty, and the associated derivation tree still open. In that case, *any* expansion of the tree is admissible. The solver then calls a standard grammar fuzzer to close the tree using random expansions (again, the number of requested solutions is configurable). This procedure ensures that the solver does not generate too many solutions that look alike by considering all possible grammar expansions in all cases.

There are two more constraint solver components, which provide solutions for *existential quantifiers* and *semantic predicates.*

Existential Quantifiers. Existential quantifiers (e.g., "there is an outer XML tag defining a given namespace") not matching the current derivation tree are eliminated using the "tree insertion" component, which searches for opportunities to insert the requested tree into the existing constrained derivation tree. The inserted tree contains a node labeled with the nonterminal type of the variable bound by the quantifier, and optionally contains subtrees for match expression elements. For the XML namespace example, the component will, e.g., *replace* an existing ⟨*xml-tree*⟩ subtree with the tree to insert, and in turn *add the replaced tree as a subtree of the inserted tree.* Tree insertion can cause violations of *already eliminated* constraints. Thus, the *original constraint is re-inserted* afterward. If none of the already solved constraints were violated, the added constraint is quickly eliminated again. Alternatively, the added CDT is discarded, or further insertions are performed

Semantic Predicates. ISLa distinguishes *structural* and *semantic* predicates. Structural predicates, like "`inside`" in Listing 2, evaluate to *true* for *false. Semantic* predicates, for which specific solvers have to be implemented, can additionally evaluate to an *assignment*, similarly to SMT formulas, or *"not ready"* if the result differs for different expansions of open argument derivation trees. We use them to address shortcomings of SMT solvers or the SMT-LIB language. Classic use cases are *checksum* predicates—encoding checksums in SMT-LIB is cumbersome at least—and structure-aware predicates like the `count` predicate used in our CSV case study, which produces rows with a specific number of columns.

Quantifiers over Integers. Existential numeric quantifiers are eliminated by introducing fresh numeric constants. Their universal counterparts are more complicated. If the core of such a formula constrains the range of the quantified variable, ISLa enumerates all possible values. Additionally, the solver implements some transformation-based approaches for formulas of specific structure.

Conjunctions, Disjunctions, Negations. The solver pushes negations inside formulas, splits conjunctions into several elements of the constraint set of one CDT, and disjunctions into several CDTs.

Cost Function. The choice of the cost function impacts the solver's performance, both in terms of efficiency (generated inputs per second) and diversity (input features covered). Our cost function computes the weighted geometric mean of different *cost factors.* We provide a sensible default weight vector. Furthermore, weights can be manually configured, and we provide an optimizer using an evolutionary algorithm for choosing good weights. We currently consider five cost factors: (1) *Tree closing cost.* We approximate the cost to close a derivation tree by the sum of the estimated instantiation effort for all leaf nonterminal symbols. (2) *Constraint cost.* This assigns higher cost to constraints that are more expensive to solve, notably existential quantifiers that have to be eliminated by tree insertion. (3) *Derivation depth.* Assigning higher costs to CDTs generated later in the process can prevent starvation of states added earlier. (4) *k-path coverage.* We use the k-path coverage metric [10] to determine the context-sensitive input feature coverage of derivation trees. We penalize trees covering only few k-paths. The concrete value of k is configurable; the default is 3. (5) *Global k-path coverage.* This factor assigns a higher cost to trees whose k–paths have already been covered by existing trees in the queue. The history of covered paths is reset once all paths have been covered.

Listing 3: ISLearn pattern obtained from Listing 2

```
forall <?NONTERMINAL> attribute="{<?MATCHEXPR(prefix_use)>}"
    in start: ((= prefix_use <?DSTRINGS>) or
  exists <?NONTERMINAL> outer_tag="{<?MATCHEXPR(cont_attribute)>}"
      in start: (inside(attribute, outer_tag) and
    exists <?NONTERMINAL> def_attribute=
        "{<?MATCHEXPR(prefix_def)>}" in cont_attribute:
      (= prefix_use prefix_def)))
```

5 MINING ISLA CONSTRAINTS

The ISLa components introduced so far enable developers to *manually* specify input constraints based on an analysis of input formats; and to use these constraints for input validation and generation. With these constraints, developers do not have to code domain-specific input generators or checkers; furthermore, ISLa constraints can easily be refined and specialized, e.g., by adding another constraint for targeted testing. Yet, the full potential of such a declarative specification language materializes when we *automatically mine* input constraints from samples and automatic experiments. This enables us to *connect any observable program behavior with constraints on system inputs*. Example behaviors of interest include normal completion, reaching some point in code, or crashing.

To that end, we developed ISLearn, a *miner for input constraints*. ISLearn is inspired by Daikon [6], a tool for learning likely unit-level program invariants from dynamic execution traces. Daikon checks for invariants from a predefined set of *patterns* (e.g., value ranges and sortedness). ISLearn also uses patterns. The main differences to Daikon are: (1) ISLearn mines conjunctions of disjunctions of *quantified*, *structure-aware* formulas; Daikon generates invariants over literals or simple collections, (2) Daikon requires a meaningful *test suite* to obtain feasible unit-level execution traces. ISLearn can *automatically generate more inputs* satisfying a program property, and *reduce* those inputs to their *essential features*, (3) ISLearn also considers *negative* inputs (not satisfying a property) to estimate the *specificity* of invariants, and (4) ISLearn can easily be *extended with more patterns* by adding them to a human-readable configuration file. All these are unique features of ISLearn.

The main inputs to the ISLearn system, apart from a *grammar* of the input language, are sets of positive and negative *sample inputs*, and a *program property* (e.g., the program terminates normally). Both are optional: Invariants can be mined from inputs only, and inputs can be automatically generated from only the property.

Patterns are defined in a superset of the ISLa language, enriched with *placeholders* for *nonterminal types* (`<?NONTERMINAL>`), *match expressions* (`<?MATCHEXPR(params)>`, where `params` is a list of variables that should be bound in the instantiated match expression, and *string constants* (`<?STRING>`). The `<?DSTRINGS>` placeholder can be instantiated by *multiple* strings; the surrounding, atomic ISLa formula is expanded to a *disjunction* for all instantiations.

Consider the constraint for prefix bindings in XML attribute from Listing 2. We abstract this constraint to an ISLearn pattern by replacing all nonterminal types and match expressions by corresponding placeholders. The constant `"xmlns"` is abstracted by a `<?DSTRINGS>` placeholder to permit instantiations by multiple keywords. The resulting pattern is shown in Listing 3.

This is not the only possible abstraction. In fact, to recover an equivalent invariant for the original XML constraint, we need to introduce another variable `ns_prefix` in the match expression for `def_attribute` for binding the constant `xmlns`, along with an equality constraint "`(= ns_prefix <?STRING>)`". However, we found that the particular pattern in Listing 3 is still useful. In our evaluation (Section 6), we applied ISLearn to languages which did not inform our pattern catalog. One of our evaluation targets is the Racket language from the Lisp family. Since Racket programs are, similarly to XML, tree structures, the first abstraction of the XML pattern can be instantiated to a definition-use invariant for Racket. The `<?DSTRINGS>` placeholder is instantiated by all functions used in the learning samples, such as `*`, `+`, and `sqrt`. The constraint `(= prefix_use <?DSTRINGS>)` gets a disjunction of equalities, one for each used function. "Def-Use" constraints for languages like C do not apply to Racket. For example, in Racket it is much more difficult to distinguish a variable from a function symbol in expressions than in C, which is why function symbols have to be explicitly excluded in the constraint. Also, definitions have to occur in an *outer* scope instead of *before* the use of a variable.

ISLearn operates in three phases. The *input augmentation phase* uses both a *grammar fuzzer* and a grammar-, property-, and k-path-aware *mutation fuzzer* to generate more input samples, both satisfying (positive) and violating (negative) the given program property (if any). The obtained inputs are optionally reduced, maintaining their relation w.r.t. the program property and the covered k-paths. Then, learning samples are selected from the positive inputs, minimizing their size while maximizing total k-path coverage.

The *candidate generation phase* instantiates selected patterns from the catalog based on the given learning inputs in several steps. For example, the first step instantiates nonterminal placeholders in quantifiers and match expression placeholder arguments. The results after each instantiation phase are *approximately filtered* using an ISLa checker for schematic formulas. The filtering is conservative: Whenever some learning input *might* satisfy a partially instantiated pattern, that pattern is retained.

Finally, the *filtering and combination phase* combines candidate invariants to *conjunctions of disjunctions* satisfying configurable *target values for recall and specificity*. First, we evaluate for each candidate which of the positive and negative inputs it satisfies. Then, we combine candidates to disjunctions up to a configurable size, such that the percentage of positive inputs satisfying the combination exceeds the recall threshold. Candidates are only combined if the recall estimate of the combination exceeds the estimate of all participants. In a next step, we combine the disjunctions to conjunctions to maximize the amount of negative inputs *not* satisfying the resulting combinations (specificity). ISLearn returns invariants with sufficient specificity and recall estimates, in descending order according to the values of these estimates.

Implementation. The ISLa core, checker, and solver, as well as ISLearn are implemented in Python.[2] We use the Z3 SMT solver, and a coverage-aware grammar fuzzer based on the Fuzzing Book [29] for closing unconstrained trees. We implemented two additional libraries for graph operations on CFGs (reachability, k-paths) and for approximating grammars by regular expressions.

[2]The ISLa prototype is available for the ESEC/FST reviewers in an anonymous repository at https://anonymous.4open.science/r/isla-esec-fse/; the ISLearn prototype is available at https://anonymous.4open.science/r/islearn-esec-fse/. We will release both under an open source license after acceptance of this paper.

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`	✗	✗	✓	✗
Racket	`racket`				
DOT	`dot`	To be mined in Section 6.3			
ICMP Echo	`pythonping`				

6 EVALUATION

To evaluate ISLa and ISLearn, we pose three research questions:

RQ1 To which degree do ISLa constraints contribute to the *efficiency* and *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 *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).

RQ3 What are the recall and specificity of invariants mined by ISLearn? We evaluate how useful the invariants mined by ISLearn, and specifically the default patterns, are to describe the circumstances of *normal* program behavior.

Evaluation Subjects. To evaluate RQ1 and RQ2, 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 (*len-cnt*).

To cover these properties, we chose input languages of 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). Scriptsize-C extends Tiny-C [7] by explicit variable declarations. For each of these languages, we defined *grammars* from their specification; for XML, we extended a pre-existing grammar from the Fuzzing Book [29] with namespace prefixes. We then added ISLa semantic constraints to all of these.

The TAR archive format represents properties of binary inputs; it 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 implemented a dedicated semantic predicate for TAR checksums (15 lines of code).

For ISLearn, we chose three *additional* languages to evaluate how well patterns from our catalog transfer to new application scenarios. Again, we aimed at choosing a diverse range of evaluation targets: (1) a *functional programming language* (Racket), (2) a *graph description language* (DOT), and (3) a *binary format* (ICMP packets).

Table 1 gives an overview of languages, test targets, and properties used in our evaluation. For the ISLearn subjects, we leave the properties open, since the goal is to *discover* their invariants. For ground truth, we chose test targets processing each language.

Table 2: ISLa Efficiency, precision, and input diversity

	Constraints	LOC	Efficiency	Precision	Diversity	Length
			Inputs/min	Inputs/min (%)	%k-paths	#Chars
C	(none)	—	504	121(24)	48	1
	+ no-redef	4	204	35(17)	52	14
	+ def-use	6	198	198(100)	66	32
XML	(none)	—	786	142(18)	80	5
	+ balance	2	2,142	2,078(97)	92	44
	+ def-use	11	60	57(95)	89	134
	+ no-redef	5	60	60(100)	88	127
TAR	(none)	—	749	0(0)	0	0
	+ length	44	35	0(0)	0	0
	+ checksum	3	30	10(33)	81	3,072
	+ reference	14	42	42(100)	80	3,072
reST	(none)	—	252	86(34)	100	9
	+ reference	6	726	428(59)	100	36
	+ length	7	462	402(87)	100	34
	+ numbering	7	498	483(97)	100	35
	+ no-redef	4	498	498(100)	100	34
CSV	(none)	—	774	472(61)	100	8
	+ columns	7	30	30(100)	100	1,699

The "Efficiency" column considers *all* produced inputs; "Precision", "Diversity" and "Length" only *valid* (accepted) inputs. "Length" is the median length of all valid inputs. We evaluated k-path coverage for both k=3 and k=4.

6.1 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 for a semantic property and the resulting overall precision. The "(none)" rows stand for "no constraint" added. Here, we ran the grammar fuzzer that ISLa uses to close unconstrained open derivation trees. For each language, the rows below "(none)" show the results of the ISLa generator when adding the specified constraint *on top* of the ones appearing above. The first constraint is the one with the most positive effect on precision; similarly for the others. The "Precision" column shows the number of *valid* inputs generated per minute, with the percentage of valid inputs in parentheses. Only 18% of generated XML inputs are valid without constraints; 142 valid XML documents are generated per minute. For TAR, not a single input is valid. The "Efficiency" column displays the generation speed irrespectively of validity. We observe that ISLa generates dozens to hundreds of inputs per minute, including a high number of valid ones. 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. For Listing 1 (*balance* in Table 2), the length is 2.

In most cases, the precision increases with each additional constraint. For XML, a total of 18 lines of constraints is required to achieve 100% precision, with the last one going from a value slightly below 100% towards the totality of all inputs. Interestingly, relative precision as well as overall efficiency decline when adding the *def-use* property to XML. The loss in efficiency can be explained by the expensiveness of the *def-use* constraint, which requires complex derivation tree manipulations. The precision loss stems from the fact that the solver is now directed towards introducing more attributes with namespace prefixes, which introduces more (invalid) attribute repetitions. The *no-redef* constraint increases precision up to 100%. A similar phenomenon can be observed for Scriptsize-C.

✏ Already with few ISLa constraints, a high number of valid inputs is generated.

✏ 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* length and checksum constraints, we cannot produce even a single valid TAR file.

6.2 RQ2: Diversity

A test generator should produce inputs exercising different language features, by which one can expect to reach different paths in the language processor [10]. 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 their *accumulated k-path coverage* [10], 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- and 4-paths* during a one-hour run per all 3/4-paths in the grammar. For example, generating Scriptsize-C programs from the grammar only achieves 48% coverage, while we cover 66% of all 3/4-paths when adding the "no-redef" and "def-use" constraints. We only count valid inputs accepted by the program under test.

Overall, the inputs produced by ISLa have *better diversity* than inputs produced without constraints.

Especially to explain the behavior of the solver for CSV, we also added the "Length" column to the table. Input length is not a particularly good coverage measure, as one can always choose, e.g., very long identifiers. However, we can observe that, in particular for CSV and C, the inputs generated by the grammar fuzzer are mostly trivial; the most common valid C program generated by the grammar fuzzer is "`;`". In general, the ISLa solver clearly outperforms the grammar fuzzer in terms of complexity of the generated inputs.

✏ ISLa covers the diversity of the underlying grammar.

6.3 RQ3: ISLearn

We populated the patterns catalog for ISLearn with abstractions of the patterns used for the ISLa evaluation targets. In addition, we added a couple of simple properties about magic constants, most notably the pattern "String Existence":

```
forall <?NONTERMINAL> container in start:
  exists <?NONTERMINAL> elem in container:
    (= elem <?STRING>)
```

The goal of this research question is to assess how well these patterns, and the ISLearn system in general, can be used to mine invariants describing circumstances of normal program behavior (i.e., whether an input is accepted by the program under test). We are particularly interested in two questions: (1) If an input is *valid* (accepted by the program), what is the probability that the mined invariant actually classifies the input as such (i.e., the ISLa checker reports that the input does not satisfy the invariant)? This is captured by the *recall* of the invariant. (2) Conversely, if an input is *in*valid, what are the chances that the mined invariant classifies it accordingly? This is assessed by the *specificity* of the invariant.

To evaluate recall and specificity, we chose seed sets of *training* and *validation* inputs. For Racket and DOT, we obtained valid Racket or DOT files from GitHub. We separated those inputs into sets of training and validation inputs of equal size. Subsequently, we expanded the training and validation sets to 50 inputs each using both a mutation-based and a grammar fuzzer. Similarly, we collect negative inputs (not accepted by the programs under test) into sets of negative training and validation inputs, each of size 50. Our third evaluation target are ICMP Echo packets (as used by the ping utility).We generated random, valid echo request and reply packets with correct checksums using the 'pythonping' library. To obtain negative samples, we created arbitrary (not necessarily echo) ICMP packets, 20% of those with an *incorrect* checksum value.

	Classified as		
Input	**True**	**False**	**Total**
True	TP = 50	FN = 0	50
False	FP = 8	TN = 42	50
Total	58	42	100

Recall = 100%, *Specificity* = 84%
Precision = 86%, *Accuracy* = 92%

(a) DOT

	Classified as		
Input	**True**	**False**	**Total**
True	TP = 50	FN = 0	50
False	FP = 3	TN = 47	50
Total	53	47	100

Recall = 100%, *Specificity* = 94%
Precision = 94%, *Accuracy* = 97%

(b) ICMP Echo

	Classified as		
Input	**True**	**False**	**Total**
True	TP = 36	FN = 14	50
False	FP = 8	TN = 42	50
Total	44	56	100

Recall = 72%, *Specificity* = 84%
Precision = 82%, *Accuracy* = 78%

(c) Racket (XML pattern)

	Classified as		
Input	**True**	**False**	**Total**
True	TP = 36	FN = 14	50
False	FP = 5	TN = 45	50
Total	41	59	100

Recall = 72%, *Specificity* = 90%
Precision = 88%, *Accuracy* = 81%

(d) Racket (XML + reST pattern)

Table 3: Confusion matrices for RQ3

Based on the (positive and negative) training examples, we let ISLearn compute an invariant. The system already estimates recall and specificity based on the supplied sample inputs, and returns the top-ranked result. We then assessed the quality of that invariant using the validation sets. If, e.g., an input from the positive validation set does *not* satisfy the invariant, the input is a *false negative* (FN).

Table 3 presents the *confusion matrices* for our evaluation. For DOT, ISLearn discovered the invariant that edges in directed graphs are directed (`->`), and undirected (`--`) for undirected graphs. The invariant is slightly too weak, as it only requires *one* correct edge in each "edge statement," which, however, can contain multiple (right or wrong) edges. In the case of ICMP Echo packets, the system learns that the value of the "type" is 0 (reply) or 8 (response). It wrongly classifies three packets with wrong checksums as valid. Adding a pattern for a semantic predicate computing *internet checksum* achieves 100% specificity. Both of these invariants are obtained from combined instantiations of the "string existence" pattern. We already mentioned that a *def-use* invariant for variables in Racket can be obtained from a pattern derived from an XML invariant; this leads to 72% recall and 84% specificity. One missing semantic feature is a *def-use* property for *functions*. We discovered that by weakening the *def-use* pattern obtained from reST, taking into account predefined function symbols that have not been defined, we obtain a suitable invariant for this property. The confusion matrix in Table 3d demonstrates that this increases specificity to 90%. The—compared to DOT and ICMP—low recall stems from the fact that not all predefined functions appear in the training set.

✏ ISLearn mines invariants of high recall and specificity based on patterns for re-occurring input properties.

6.4 Threats 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 and ISLearn 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. This supports the claim that those are representative and can be transferred to different targets, as does the fact that an XML pattern could be used for Racket.

7 RELATED WORK

Parser Specifications. ISLa provides a framework to specify *input requirements*, or preconditions, of a program. It targets the system level, where inputs are generally strings. *Parser generators* like ANTLR[3] and the pioneer yacc [15] promoted CFGs for specifying complex structured inputs. However, specifications designed for *parsing* inputs are rarely specific enough to also be used for *producing* valid inputs, which is the gap that ISLa fills.

Attribute Grammars. Attribute grammars [16] associate grammar symbols with synthesized and derived attributes. This allows checking semantic properties; if attributes use a general-purpose programming language, one can express arbitrarily complex semantic properties. The meta-compiler JastAdd [11], for instance, supports imperative specifications in Java; 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.

Grammar-Based Test Generation. Context-Free Grammars are well-suited for syntax-aware test input generation. CSmith [27] and LangFuzz [13] use CFGs as a basis to randomly create syntactically valid C and JavaScript programs, respectively; Grammarinator [12] produces inputs from ANTLR grammars. The underlying grammars are typically hand-written, but can also be *mined* from programs [9] and inputs [17]. ISLa fits 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 a valid TAR file from a CFG approaches zero, and CSmith can only generate—well—C files.

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 and complex constraints, such as *def-use* properties, have to be programmatically implemented. 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. [21] use Higher-Order Attribute Grammars [26] for fuzz testing, providing custom predicates for parse tree manipulation (e.g., length constraints and checksum computation) in a general-purpose programming language. 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. As Prolog is relatively declarative, CLP shares the idea of declarative constraints with ISLa; however, CLP-based language fuzzers also cannot be used for parsing.

Property-Based Testing. Pioneered by QuickCheck [2], Property-Based Testing (PBT) automatically produces *data structures of the host language* to test individual functions against user-defined properties. This allows expressing features in the host programming language, which is not available when working with unstructured system inputs. ProSyT [3] and Luck [18] generate data structures for Erlang and Haskell, respectively, separating and solving semantic constraints from data types.

Generally, the concept of *parsing* and *mutating* existing data is not present in PBT. One exception is Zest [20], which leverages program feedback to create syntactically valid input mutants exercising interesting program paths. 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.

Mining Invariants. Daikon [6] is the seminal work for extracting *invariant candidates* from program executions—pre- and postconditions as well as data invariants; its pattern matching approach is the inspiration for ISLearn. Recent advances in the field focus on program verification, loop invariants, and the usage of neural networks [28]. Unlike ISLearn, all these approaches operate at the unit level, and cannot generate targeted executions to refine invariants. To the best of our knowledge, ISLearn is the first approach to specify, determine, and refine invariants at the system level.

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 and semantics, and demonstrated that our ISLa solver can be used to generate semantically correct inputs significantly faster than by generating from a CFG alone. Furthermore, we introduced the ISLearn input invariant miner, which automatically produces useful ISLa specifications based on a program property and/or sample inputs.

Besides further refining the ISLa and ISLearn implementation, our future work will focus on the following topics:

Fuzzer integration. ISLa-generated inputs can serve as high-quality *seed inputs* for greybox fuzzers like AFL; ISLa's checkers can quickly filter out invalid generated inputs.

Testing strategies. A probabilistic variant of ISLearn could quickly *learn* which input features *correlate* with program behaviors

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

(including failures or specific coverage); this allows for test generation techniques exploring syntax and semantics.

Constraint synthesis. Besides checking *patterns*, techniques from *program synthesis* would have great potential for generating constraints from examples.

Constraints as oracles. As ISLa allows extracting and assessing arbitrary input elements, it can also check *outputs* for constraints. This allows using ISLa constraints as *oracles* (which could also be learned via ISLearn).

Detecting anomalies. Decomposing inputs and outputs provides plenty of syntactical and semantic *features* that can be used for learning commonalities and anomalies; learned correlations can be reinforced by ISLa-generated tests.

The ISLa and ISLearn prototypes are available at

https://github.com/rindPHI/isla
https://github.com/rindPHI/islearn

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] Rahul Gopinath, Björn Mathis, and Andreas Zeller. 2020. Mining Input Grammars from Dynamic Control Flow. In *Proceedings 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FST)*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 172–183. https://doi.org/10.1145/3368089.3409679

[10] 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

[11] 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

[12] 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

[13] 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

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

[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] Neil Kulkarni, Caroline Lemieux, and Koushik Sen. 2021. Learning Highly Recursive Input Grammars. In *36th IEEE/ACM International Conference on Automated Software Engineering.* ACM New York, NY, USA.

[18] 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

[19] 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

[20] 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

[21] 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

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

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

[24] 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.

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

[26] 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

[27] 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

[28] Jianan Yao, Gabriel Ryan, Justin Wong, Suman Jana, and Ronghui Gu. 2020. Learning Nonlinear Loop Invariants with Gated Continuous Logic Networks. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 106–120. https://doi.org/10.1145/3385412.3385986

[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.

A APPENDIX

In this appendix, we provide additional examples, formal definitions, theorems, and proof sketches. Furthermore, we show the invariants that ISLearn mined in our evaluation (RQ3).

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. 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:

```
forall <section-title> title=
    "{<title-txt> titletxt}\n{<underline> 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.

```
forall <header> header in start:
  forall <checksum> 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 occurrences of some nonterminal in an input tree, and fixes trees with an insufficient number of occurrences if possible. The following ISLa constraint for the CSV property uses an additional language feature: It introduces a *numeric constant* `colno` using the **`num`** directive, which works similarly to let expressions in functional programming languages. It is primarily—and also in this example—used to enable information exchange between semantic predicate formulas.

```
forall <csv-header> hline in start:
  exists int colno:
    ((>= (str.to_int colno) 3) and
    ((<= (str.to_int colno) 5) and
     (count(hline, "<raw-field>", colno) and
     forall <csv-record> line in start:
       count(line, "<raw-field>", colno))))
```

One has to be aware that the *order* of semantic predicates in a constraint matters. This is in contrast to all other language atoms: SMT formulas, in particular, are fed to an SMT solver only after all universal quantifiers have been eliminated resp. matched, and evaluated *en bloc.* Semantic predicates, on the other hand, are generally not compositional. When computing the checksum for a TAR file, for instance, it is important that all elements of the file header are already fixed at that point, i.e, all semantic predicates on header elements have to be evaluated before. Consequently, they have to occur before the checksum predicate in the overall constraint. Despite this particularity, semantic predicates are an easy way to increase both the expressiveness and solving performance of ISLa constraints, and to overcome the limits of SMT-LIB and off-the-shelf solvers.

3 ISLa Syntax and Semantics

We provide a more formal 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 $\lessdot$.

Definition A.1 (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 $\lessdot_S$ 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 $v \lessdot_V v_i$ and $v_i <_S v_l$ 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. We define $closed(t) :\equiv \forall l \in leaves(t) : l \in T$, and $open(t) :\equiv \neg closed(t)$. $\mathcal{T}(G)$ is the set of all (closed and open) derivation trees for G.

Example A.2. We explain the formal definition of derivation trees (Definition A.1) along the XML document `<a>x</a>`. visualized in Fig. 2. 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

Listing 4: 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
```

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 $v_6 \lessdot_S v_8$ (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-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-open-tag*$\rangle$, '<', $\langle$*id*$\rangle$, '>') in the grammar, which is indeed the case, since '<' $\langle$*id*$\rangle$ '>' is an expansion alternative (the first one) for the nonterminal $\langle$*xml-open-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).

We explain the process of *matching quantified formulas with match expressions against derivation trees* along the concrete code from our implementation (Listing 4). The function `match` receives a derivation tree, a match expression, a map for storing the result, and a list of matches to ignore (which 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 tree $t \in \mathcal{T}(G)$ and match expression $\mathit{mexpr} \in (N \times T \times \mathrm{VSym})^*$, leaving the conversion of symbols from N and T to variables, flattening, and backtracking implicit. The return type of this function is an optional mapping of variables to subtrees. We say that *there is a match m* for t and *mexpr* if *match* returns a mapping (and not **`None`**).

4 Solving ISLa Constraints

We provide a formalization of our ISLa constraint solver, including two correctness theorems and proof sketches.

We formalize input generation for ISLa as a transition system between *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

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

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

$$\{\textbf{forall}\ \langle id\rangle\ id\ \textbf{in}\ \mathit{1}\texttt{: (= (str.len}\ id\texttt{) 17)}\} \triangleright t.$$

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 A.3 (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

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

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 3.6 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 3.8:

Definition A.4 (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

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

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

Definition A.5 (CDT Transition System). A *CDTTS* for a signature $\Sigma = (G, \text{PSym}, \text{VSym})$ is a transition system $(C, \longrightarrow)$, where, for $\Phi \in 2^{\text{Fml}}$ and $t \in \mathcal{T}(G)$, C consists of CDTs $\Phi \triangleright t$, and $\longrightarrow \subseteq C \times C$. We write $\mathit{cdt} \longrightarrow \mathit{cdt}'$ if $(\mathit{cdt}, \mathit{cdt}') \in' \longrightarrow$.

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 an 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 A.6 (Global Precision and Completeness). Let $(C, \longrightarrow)$ be a CDTTS, and R_{cdt} be the set of all closed trees t such that $\mathit{cdt} \longrightarrow \cdots \longrightarrow \emptyset \triangleright t$ is a derivation in $(C, \longrightarrow)$. Then, $(C, \longrightarrow)$ is *globally precise* if, for each CDT cdt in the domain of $\longrightarrow$, it holds that $[\![\mathit{cdt}]\!] \supseteq \{\mathit{str}(t) | t \in R_{cdt}\}$. The CDTTS is *globally complete* if it holds that $[\![\mathit{cdt}]\!] \subseteq \{\mathit{str}(t) | t \in R_{cdt}\}$.

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 A.7 (Local Precision and Completeness). A CDTTS $(C, \longrightarrow)$ is *precise* if, for each CDT cdt in the domain of $\longrightarrow$, it holds that $[\![\mathit{cdt}]\!] \supseteq \bigcup_{\mathit{cdt} \to \mathit{cdt}'}([\![\mathit{cdt}']\!])$. The CDTTS is *complete* if it holds that $[\![\mathit{cdt}]\!] \subseteq \bigcup_{\mathit{cdt} \to \mathit{cdt}'}([\![\mathit{cdt}']\!])$.

As for "soundness" in first-order logic (see, e.g., [25]), 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 A.8. 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 A.8. *A locally precise CDTTS is also globally precise.*

Proof. The lemma trivially holds if $R_{cdt} = \emptyset$. Otherwise, let $\mathit{cdt}_0 \longrightarrow \mathit{cdt}_1 \longrightarrow \cdots \longrightarrow \emptyset \triangleright t$ be any transition chain s.t. $\mathit{cdt}_0 = \mathit{cdt}$ and $t \in R_{cdt}$. Then, it follows from local precision that $[\![\mathit{cdt}_k]\!] \supseteq [\![\mathit{cdt}_{k+1}]\!]$ for $k = 0, \ldots, n-1$, and by transitivity of $\supseteq$ also $[\![\mathit{cdt}_k]\!] \supseteq [\![\mathit{cdt}_l]\!]$ for $0 \leq k < l \leq n$. Since $[\![\emptyset \triangleright t']\!] = \mathit{str}(t')$ 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. 4. 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 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 are returned if the given tree can be completed or "fixed" to a satisfying solution. One may obtain

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

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

$$\begin{aligned}&\{\ldots, \texttt{forall int}\ n\ \texttt{in}\ \varphi\colon, \ldots\} \triangleright^{I} t \;\rightarrow\\ &\quad \{\ldots, \{n \mapsto c\}(\varphi), \ldots\} \triangleright^{I} t\\ &\quad \text{where } c \in \mathrm{VSym} \text{ is } \textit{fresh} \text{ and } vtype(c) = \texttt{int}\end{aligned} \tag{3}$$

$$\begin{aligned}&\{\ldots, \overbrace{\texttt{forall}\ type\ v\ \texttt{in}\ id\colon \varphi}^{\psi}, \ldots\} \triangleright^{I} t \;\rightarrow\; \{\{\ldots,\ldots\} \triangleright^{I} t\}\\ &\quad \text{if } \forall \text{ subrees } t' \text{ of } \mathcal{T}(\beta_t(id)):\\ &\quad (\psi, t') \in I \vee label(t') \neq type\end{aligned} \tag{4}$$

$$\begin{aligned}&\{\ldots, \overbrace{\texttt{forall}\ type\ v\text{=``}mexpr\text{''}\ \texttt{in}\ id\colon \varphi}^{\psi}, \ldots\} \triangleright^{I} t \;\rightarrow\\ &\quad \{\{\ldots,\ldots\} \triangleright^{I} t\} \text{ if } \forall \text{ subrees } t' \text{ of } \mathcal{T}(\beta_t(id)):\\ &\qquad (\psi, t') \in I \vee label(t') \neq type \vee\\ &\qquad \text{there is no } m = match(t', mexpr)\end{aligned} \tag{5}$$

$$\begin{aligned}&\{\ldots, \overbrace{\texttt{forall}\ type\ v\ \texttt{in}\ id\colon \varphi}^{\psi}, \ldots\} \triangleright^{I} t \;\rightarrow\\ &\quad \Big\{\{\ldots,\psi,\ldots\} \cup \bigcup \Psi \triangleright^{I \cup (\{\psi\} \times T)} t\Big\} \text{ where}\\ &\quad \Psi = \big\{\beta_t[v \mapsto t'](\varphi) \mid t' \in T\big\} \wedge\\ &\quad T = \big\{t' \mid t' = \beta_t(id) \wedge (\psi, t') \notin I \wedge\\ &\qquad\qquad label(t') = type\big\} \neq \emptyset\end{aligned} \tag{6}$$

$$\begin{aligned}&\{\ldots, \overbrace{\texttt{forall}\ type\ v\text{=``}mexpr\text{''}\ \texttt{in}\ id\colon \varphi}^{\psi}, \ldots\} \triangleright^{I} t \;\rightarrow\\ &\quad \Big\{\{\ldots,\psi,\ldots\} \cup \bigcup \Psi \triangleright^{I \cup (\{\psi\} \times T)} t\Big\} \text{ where}\\ &\quad \Psi = \big\{\beta_t[v \mapsto t'][m](\varphi) \mid (t', m) \in T\big\} \wedge\\ &\quad T = \big\{(t', m) \mid t' = \beta_t(id) \wedge (\psi, t') \notin I \wedge\\ &\qquad label(t') = type \wedge\\ &\qquad \text{there is an } m = match(t, mexpr)\big\} \neq \emptyset\end{aligned} \tag{7}$$

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

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

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

$$\begin{aligned}&\{\ldots, \overbrace{\texttt{exists}\ type\ v\ \texttt{in}\ id\colon \varphi}^{\psi}, \ldots\} \triangleright^{I} t \;\rightarrow\\ &\quad \bigcup_{\xi \in \Psi} \big\{\{\ldots,\xi,\ldots\} \triangleright^{I} t\big\} \text{ where}\\ &\quad \Psi = \big\{\beta_t[v \mapsto t'](\varphi) \mid t' \in T\big\} \wedge\\ &\quad T = \big\{t' \mid t' = \beta_t(id) \wedge (\psi, t') \notin I \wedge\\ &\qquad label(t') = type\big\} \neq \emptyset\end{aligned} \tag{11}$$

$$\begin{aligned}&\{\ldots, \overbrace{\texttt{exists}\ type\ v\text{=``}mexpr\text{''}\ \texttt{in}\ id\colon \varphi}^{\psi}, \ldots\} \triangleright^{I} t \;\rightarrow\\ &\quad \bigcup_{\xi \in \Psi} \big\{\{\ldots,\xi,\ldots\} \triangleright^{I} t\big\} \text{ where}\\ &\quad \Psi = \big\{\beta_t[v \mapsto t'][m](\varphi) \mid (t', m) \in T\big\} \wedge\\ &\quad T = \big\{(t', m) \mid t' = \beta_t(id) \wedge (\psi, t') \notin I \wedge\\ &\qquad label(t') = type \wedge\\ &\qquad \text{there is an } m = match(t, mexpr)\big\} \neq \emptyset\end{aligned} \tag{12}$$

$$\begin{aligned}&\{\ldots, \texttt{exists}\ type\ v\ \texttt{in}\ id\colon \varphi, \ldots\} \triangleright^{I} t \;\rightarrow\\ &\quad \big\{\{\ldots, \{v \mapsto nid\}(\varphi), \ldots\} \cup \Phi_{orig} \triangleright^{I} \{id \mapsto t'\}(t) \mid\\ &\quad (nid, t') \in insert(makeTree(v), \beta_t(id))\end{aligned} \tag{13}$$

$$\begin{aligned}&\{\ldots, \texttt{exists}\ type\ v\text{=``}mexpr\text{''}\ \texttt{in}\ id\colon \varphi, \ldots\} \triangleright^{I} t \;\rightarrow\\ &\quad \big\{\{\ldots, \{v \mapsto nid\}(\varphi), \ldots\} \cup \Phi_{orig} \triangleright^{I} \{id \mapsto t'\}(t) \mid\\ &\quad (nid, t') \in insert(makeTree(v, mexpr), \beta_t(id))\end{aligned} \tag{14}$$

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

$$\begin{aligned}&\Phi \triangleright^{I} t \;\rightarrow\; \{\Phi \triangleright^{I} t' \mid t' \in \mathcal{T}(t)\} \neq \{\Phi \triangleright^{I} t\}\\ &\quad \text{if } \Phi \subseteq \{p(\ldots) \mid p \in \mathrm{PSym}_{sem}\}\end{aligned} \tag{16}$$

Figure 4: Efficient ISLa CDTTS Transition Relation

$\circlearrowleft$ 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. 4. Rule (1) uses a function $inv : \mathrm{Fml} \rightarrow 2^{\mathrm{Fml}}$ 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. Existential quantifiers over numbers are eliminated in Rule (3) by introducing a fresh (not occurring in the containing CDT) variable symbol with the special nonterminal type `int` 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 $expand_{\Phi}^{\forall}(t)$ 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 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 A.9. *(Precision) The ISLa CDTTS in Fig. 4 is globally precise.*

Proof Sketch. By Lemma A.8, we prove global precision by showing that each individual transition rule is *locally* precise, i.e., that the produced states do not represent derivation trees that were not originally in the semantics of the inputs CDT.

Rule (1) is precise since conversion to Disjunctive Normal Form is equivalence-preserving. Elimination of structural predicates (Rule (2)) is trivially precise (removing a true element from a conjunction does not change the semantics).

Rules (4) and (5) are precise because a universal quantifier that does not match any tree element evaluates to true according to Definition 3.6, and we only remove it if we can be sure that no possible extension of an open tree will ever match the quantifier. If a match is already in the index set, we can be sure that it already has been considered due to the definition of Rules (6) and (7), which are the only rules ever adding to that set.

Rules (6) and (7) are precise because we only *add* the matching instantiations of the inner formula to the (conjunctive) constraint set.

Tree expansion (Rule (8)) is precise since by considering *more concrete* trees, the set of concrete trees represented by the input CDT is only ever *decreased* in the outputs (cf. Definition A.3).

The elimination of SMT formulas (Rule (9)) is precise since their semantics is defined via the interpretation function σ, which we query to produce valid output states. The same holds for Rule (10) for semantic predicates.

Existential quantifier matching (Rules (11) and (12)) is precise since it conforms to Definition 3.6 inasmuch it creates one instantiated CDT for each match in the input CDT. The original formula is removed from these results, but the instantiation retained.

The tree insertion rules (Rules (13) and (14)) (the most complicated ones in our system due to the complexity of tree 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 A.10. *(Completeness) The ISLa CDTTS in Fig. 4 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, although our actual implementation can only ever consider a finite set of solutions.

The same holds for solving SMT formulas. Note that if we only consider a finite solution set as in our prototype, it is crucial that *there remain no universal quantifiers* in the constraint set. Otherwise, we could obtain instantiations that conflict with formulas obtained from later quantifier instantiation. This is not a problem in the theoretic framework, though, since there we consider *all possible* solutions, of which at least some will not conflict with atoms nested in remaining universal quantifiers.

Tree insertion, which is easy to show precise, is more problematic to show locally complete, since we add the original constraint set. However, since we consider *all* possible insertions, there have to be some satisfying that constraint, since the input CDT is valid.

Since we defined one rule for each syntactic construct in ISLa, there is one rule for each valid input state. Rule (8), for example, only expands nonterminals with potential concrete subtrees matching existing existential quantifiers; for all other nonterminals, the finishing rules will be applicable.

The general measure to show that each transition produces a state that is closer to an empty constraint set is the size of the constraint set together with the nesting depth of contained quantifiers. If either of these measures decreases in each step, we eventually reach an empty constraint set. That we get closer to an empty constraint set is clear to see for all elimination rules. In case of the matching rules, we reduce the complexity of the constraint set by peeling off the outer quantifier. Again, tree insertion is most problematic: It peels of the existential quantifier, but adds the original constraint set. Here, it is important to see that there are some insertions for which we can remove the existential constraint we eliminated by tree insertion by *matching*, and that we thus do not have to keep re-inserting. □

We explain the main building blocks used in our ISLa CDTTS (Fig. 4) in more detail.

Tracking Instantiations. Our CDTTS stepwise expands open trees and checks if existing universal quantifiers match the expanded tree. Expansion does not eliminate a universal quantifier, since it might apply to not yet generated subtrees. To avoid endlessly instantiating universal quantifiers with the *same* trees, we track already performed universal quantifier instantiations. To that end, we augment CDTs with an *index set I* consisting of pairs of universal formulas and trees with which they already have been unified; we write $\Phi \triangleright^I t$ for the enhanced structures.

Invariant. We maintain the invariant that all formulas $\varphi \in \Phi$ in CDTTSs $\Phi \triangleright t$ are in NNF, i.e., negations only occur directly before predicate formulas and within SMT expressions, and are free of conjunctions and disjunctions (on top level; they are allowed inside of quantifiers and within SMT formulas). The function $inv : \mathrm{Fml} \rightarrow 2^{\mathrm{Fml}}$ first converts its argument into NNF by pushing negations inside (e.g., **not exists** *type* v **in** w: φ gets **forall** *type* v **in** w: **not** φ, and, for $\psi \in \mathrm{Trm}_{bool}(V)$, **not** ψ gets (`not` ψ) $\in \mathrm{Trm}_{bool}(V)$). Then, it converts the result into Disjunctive Normal Form by applying distributivity laws, which yields a *set* of disjunction-free formulas in NNF. Finally, it splits all top-level conjunctions outside SMT expressions in the result set into multiple formulas.

SMT Models. In Fig. 4, we apply the interpretation σ for SMT expressions to Boolean terms $\mathrm{Trm}_{bool}(V)$ with a non-empty variable set V, i.e., the evaluated expressions may contain uninterpreted String constants. In this case, the SMT solver will *either* return $\bot$ in case of an unsatisfiable constraint (or time out, which we interpret as $\bot$), *or* an assignment β (a *model*). Since we can call the solver repeatedly and ask for different solutions (by adding the negated previous 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` ⟨*xml-open-tag*⟩ `optag in tree: (= optag "<a>")`, where 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. Otherwise, we observe that the nonterminal ⟨*xml-tree*⟩ is *reachable from itself* in the grammar graph. Thus, we can replace an existing ⟨*xml-tree*⟩ node in *tree* by a number of possible alternatives, comprising <a> ⟨*xml-tree*⟩⟨*xml-close-tag*⟩, which allows to insert both the already existing ⟨*xml-tree*⟩ and the new opening tag <a> into the expanded result.

Cost Function. The choice of the right cost function is crucial for the performance of the solver, both in terms of generation speed (number of outputs per time) and output diversity (e.g., creation of deep nestings in the case of XML, or coverage of combinations of language constructs in the case of C).

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 [10] 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.

6 Evaluation

6.3 RQ3: ISLearn. In the subsequent Listings 5 to 9, we list the constraints that ISLearn mined in our case study for our third research question.

Listing 5: Constraint mined by ISLearn for DOT

```
((forall <graph_type> container in start:
    exists <DIGRAPH> elem in container:
      (= elem "digraph") or
 forall <edge_stmt> container_0 in start:
   exists <edgeop> elem_0 in container_0:
     (= elem_0 "--")) and
(forall <graph> container_1 in start:
   exists <GRAPH> elem_1 in container_1:
     (= elem_1 "graph") or
forall <edge_stmt> container_2 in start:
  exists <edgeop> elem_2 in container_2:
    (= elem_2 "->")))
```

Listing 6: Constraint mined by ISLearn for Racket based on the XML *def-use* pattern for prefixes in attributes

```
(forall <expr> attribute=
    "<maybe_comments><MWSS>{<name> prefix_use}" in start:
  ((= prefix_use "sqrt") or
   (= prefix_use "string-append") or
   ... or
  exists <definition> outer_tag="(<MWSS>define<MWSS>(<MWSS><
       ↪ name>{<WSS_NAMES> cont_attribute}<MWSS>)<MWSS><
       ↪ expr><MWSS>)"
      in start:
    (inside(attribute, outer_tag) and
    exists <NAME> def_attribute="{<NAME_CHARS> prefix_def}"
         ↪ in cont_attribute:
      (= prefix_use prefix_def))))
```

Listing 7: Racket constraint based on the XML *def-use* pattern in addition to an extended reST *def-use* pattern

```
(forall <expr> use_ctx="<maybe_comments><MWSS>(<MWSS>{<name>
     ↪  use}<wss_exprs><MWSS>)" in start:
  ((= use "sqrt") or
   (= use "string-append") or
   ... or
```

```
    exists <definition> def_ctx=
        "(<MWSS>define<MWSS>(<MWSS>{<name> def}<WSS_NAMES><
            ↪ MWSS>)<MWSS><expr><MWSS>)" in start:
      ((before(def_ctx, use_ctx) and
      (= use def))))) and
(forall <expr> attribute=
    "<maybe_comments><MWSS>{<name> prefix_use}" in start:
  ((= prefix_use "sqrt") or
   (= prefix_use "string-append") or
  ... or
  exists <definition> outer_tag="(<MWSS>define<MWSS>(<MWSS><
      ↪ name>{<WSS_NAMES> cont_attribute}<MWSS>)<MWSS><
      ↪ expr><MWSS>)" in start:
    (inside(attribute, outer_tag) and
    exists <NAME> def_attribute="{<NAME_CHARS> prefix_def}"
         ↪ in cont_attribute:
      (= prefix_use prefix_def))))
```

Listing 8: ISLearn constraint for ICMP Echo type fields

```
(forall <icmp_message> container in start:
   exists <type> elem in container:
     (= elem "00 ") or
forall <icmp_message> container_0 in start:
  exists <type> elem_0 in container_0:
    (= elem_0 "08 ")))
```

Listing 9: Constraint learned by ISLearn for ICMP Echo after adding a predicate for Internet Checksums

```
((forall <icmp_message> container in start:
    exists <type> elem in container:
      (= elem "00 ") or
 forall <icmp_message> container_0 in start:
   exists <type> elem_0 in container_0:
     (= elem_0 "08 ")))) and
forall <icmp_message> container in start:
  exists <checksum> checksum in container:
    internet_checksum(container, checksum)
```