Input Invariants in Fuzz-testing

MARKO VASYLENKO, University of Twente, The Netherlands

Fuzz-testing is a technique in which test inputs are generated programmatically to enhance software testing efficiency. This study investigates the suitability of ISLa, a declarative specification language to improve fuzz-testing. ISLa augments context-free grammars (CFG) with additional constraints to express context-sensitive input invariants. In the present study, a testing setup is developed, in which ISLa is used to specify invariants of valid test inputs, as well as test-case-specific preconditions. The expressiveness of ISLa as a specification language is evaluated, as well as the ISLa's effectiveness in generating test inputs which reveal implementation bugs. Ultimately, it is concluded that while ISLa helps tackle some challenges in generating test input, it has some fundamental and practical limitations that prevent it from being widely applicable as an input generation tool.

Additional Key Words and Phrases: property-based testing, preconditions, input fuzzing, context-free grammar

1 INTRODUCTION

Fuzz-testing is a technique used for efficient and extensive testing of software systems by means of programmatically generating test inputs. By those means, it aims to reduce the need for manual specification and introduce unusual inputs that could have been overlooked by a human tester [8]. However, test input generation might be complicated by the system under test expecting a complex, well-structured input, which is tackled in a variety of ways, e.g. by grammar-based fuzzers restricting the input structure with Context-Free Grammars (CFG) [8]. Moreover, it is common to aim at generating input that improves code coverage and explores more execution branches of the system under test [21].

Similarly to fuzz-testing, in *property-based testing* inputs are generated to satisfy some test-case-specific preconditions, to control which parts and properties of the program are tested with each test case. Notably, generating precondition-satisfying test inputs with space precondition proves challenging [5, 11, 12] and has been attempted in a variety of ways [6, 10–12]. Coverage-guided fuzzing has also been successfully used to this end [11].

The present study combines ideas of fuzz-testing and property-based testing and attempts to implement them using ISLa [15]. In the landscape of fuzzing tools, ISLa is a declarative specification language that enhances the CFG of the input language with additional *constraints*, which express potentially context-sensitive structural and semantic invariants of a valid test input. ISLa comes with ISLearn, a tool capable of mining said constraints from samples of conforming and non-conforming inputs and/or an observable system which accepts the inputs [15]. On the one hand, it has been shown to effectively fuzz inputs in languages such as XML, reST, and scriptsize-C) [15]. On the other hand, it has been speculated that ISLa could be used to express test-case specific preconditions and generate conforming test inputs [5], while ISLearn could be used to infer such preconditions from runtime observations of the system under test [15]. This paper explores the application of ISLa and ISLearn to fuzz-testing a program resembling a simple automated build system with its own configuration format, aiming to both fuzz structurally and semantically valid test inputs, and generate test inputs which trigger specific behaviours of the program.

TScIT 41, July 8, 2022, Enschede, The Netherlands

2 CHALLENGE

2.1 Input grammars and invariants

When performing fuzz-testing, valid input generation can be challenging in cases of the system under test imposing complex requirements on the input and the test cases with sparse preconditions. In particular, grammar-based systems use inputs that conform to a specific CFG for a range of purposes from modelling the problem domain to data representation [13]. Oftentimes, the input language of the system is subject to semantic and structural constraints that are inherently context-sensitive, and therefore cannot be described by a CFG, e.g. variable definition and use with block scoping in C-like languages. To that end, ISLa has been developed [15].

Listing 1. Example ISLa constraint for C definition-use semantics

```
1 forall <expr> expr in start:
2   forall <id> use_id in expr:
3     exists <declaration> decl="int {<id> def_id
          }[ = <expr>];" in start:
4       (level("GE", "<block>", decl, expr) and (
            before(decl, expr) and (= use_id
            def_id)))
```

In Listing 1, nonterminals of the target grammar are written in angle brackets. ISLa features quantifiers, like `for <expr> expr in start:` on the first line, which means that the following predicate must hold for all instances of nonterminal `<expr>` in the `start` subtree (in this case, the root of the derivation tree, quantification within any subtree bound to a variable name is possible). The quantifier also binds the subtree with `<expr>` at its root to variable `expr`, which can be later referenced in the predicate. While `forall` is a universal quantifier, ISLa also features an existential `exists`, as seen on the line 3 which quantifies the existence of at least one such derivation of `<declaration>` that matches the derivation tree pattern `"int {<id> def_id}[ = <expr>];"`. The pattern specifies the matched derivation tree of the quantified nonterminal and allows binding subtrees to variables with the `{<id> def_id}` syntax and including optional parts of the pattern in square brackets, such as `[ = <expr>]`. Finally, nested under the three levels of quantifiers is the predicate, expressed as a boolean combination of ISLa-specific predicates like `level("GE", "<block>", decl, expr)` and SMT-LIB expressions, such as `(= use_id def_id)`. This predicate, which should hold for all combinations produced by quantifiers above, expresses that there must be a declaration at the same or a higher block space level as the expression (expressed by the `level` predicate),

with the declaration occurring before the expression, where the variable name declared (`def_id`) matches the variable name used (`use_id`). In this declarative manner, ISLa is capable of expressing context-sensitive aspects of the input language structure [15].

2.2 Test preconditions

In addition to the challenge of generating valid inputs, the test cases might require inputs satisfying *sparse preconditions*, i.e. semantic invariants that do not follow directly from the data structure [11]. For example, if a procedure on a large graph has a bug that only results in incorrect input in the graph containing a specific structure, such as a cycle, a generator of arbitrary graph instances might not enumerate enough examples in the input space to find the graph with the critical structure. In this situation, the property-based testing system is unlikely to find counterexamples, despite the presence of a bug. To that end, the present study explores the use of ISLearn, a tool developed on top of ISLa for extracting ISLa constraints given a set of examples or simply a property that discriminates positive and negative examples [15]. It is done by matching patterns from ISLearn's pattern catalogue over the provided or generated sample inputs. In addition to learning context-sensitive constraints of XML, reST, DOT, scriptsize-C and other languages, it has been speculated by the authors that ISLearn can be used to extract test input generation strategies by learning constraints that correlate to certain observable program behaviours [15]. Provided that the program states of interest, e.g. reaching a certain execution branch, raising a runtime exception, or completing successfully, within each test case can be observed at runtime, these observations can be used as the property function for ISLearn to learn the ISLa constraints which correspond to the relevant test case preconditions.

2.3 Research Questions

The present study is focused on evaluating the suitability of ISLa and ISLearn for fuzz-testing, specifically, using ISLa to formalise the input language specification, using ISLearn to extract test input generation strategies from an observable system under test, and using ISLa to further generate, ideally, valid and precondition-satisfying input.

RQ1 *How well can testing preconditions be expressed in ISLa?* Is ISLa expressive enough to describe useful test preconditions? What are the fundamental limitations to its expressiveness as a specification language? Which practical limitations complicate the usage?

RQ2 *How effective is ISLa at generating test inputs?* Do generated inputs reveal bugs?

RQ3 *What (kinds of) preconditions can be mined with ISLearn?* ISLearn might not be able to pick up on all the preconditions: what separates the ones it learns from the ones it misses? Which fundamental or practical limitations hinder precondition mining?

Ultimately, we aim to evaluate how suitable ISLa in its current state is for use in fuzz-testing, and which problems or limitations affect it and might have to be tackled to improve the usability in this use case.

3 METHODOLOGY

3.1 Case study: an input language

ISLa and ISLearn will be evaluated based on a case study on an XML-based configuration language (hereafter referred to as AIL, short for An Input Language) which borrows its semantics from a few different configuration languages.

Listing 2. Example of an input language sentence

```
<build>
  <task id="main_task" main="true">
    <dep id="sub_task"></dep>
    <step cost="12">sudo rm -rf</step>
    <step script="./step1.sh" cost="30"></step>
  </task>

  <task id="sub_task">
    <step cost="1">echo hello</step>
  </task>
</build>
```

AIL is used for a simple system which resembles a simplified variation of GNU Make [4]. The language includes several constraints: the permissible tags are limited to `<build>` (only appears once, at the root of the XML tree), `<task>`, `<dep>`, and `<step>`; each `<task>` must have a **unique** `id` attribute; exactly one task must have `main="true"` to designate it as the primary build target; `<task>` appears only at the top level; within a `<task>`, there can be `<dep>` and `<step>` elements; `<dep>` must contain no inner text or sub-elements and reference an **existing** task ID with its `id` attribute; and `<step>` must have a `cost` attribute with an **integer** value and either a script attribute or content representing a command.

These constraints borrowed from formats including HTML [20], Docker Compose [2], and GNU Make [4] and are commonly featured in other configuration formats and domain-specific languages. The input language was designed to be simple yet to include several context-free and context-sensitive conditions commonly occurring in other input languages.

3.2 Conditions for tests

Taking XML (based on The Fuzzing Book [24]) as a basis, there are several additional conditions for a valid AIL sentence, which can be divided into context-free (those that can be formalised with a CFG) and, conversely, context-sensitive.

The context-free conditions considered in this case study include:

(CF1) Only correct tags are used;
(CF2) Only correct attribute names are used;
(CF3) The root of the document is the only `<build>` tag;
(CF4) Only the `<step>` tag may contain text;
(CF5) `<step>` tags only appear inside `<task>` tags;
(CF6) `<dep>` tags only appear inside `<task>` tags;

Context-sensitive conditions include:

(CS1) Task IDs must be unique;
(CS2) Dependencies should only refer to existing tasks (IDs must match);
(CS3) There must be one and only one main entry point;

(CS4) There *should* be no circular dependencies between tasks;
(CS5) All tasks *should* be in the dependency tree of the main entry point.

3.3 Fuzzing AIL tests with ISLa

The first part of the case study entails attempting to generate precondition-satisfying test inputs with ISLa, preconditions being the context-free and context-sensitive conditions described in subsection 3.2, as well as their negations. First, implementation of the separate conditions as ISLa constraints are attempted, after which they can be combined using conjunction, disjunction, and negation. While it is expected that ISLa is sufficiently expressive for most conditions, some might be impossible to translate, possibly revealing fundamental limitations of ISLa as a specification language. More subjective observations are made concerning practicalities such as the ease (or difficulty) of troubleshooting a specification.

Afterwards, fuzzing test inputs with ISLa using previously implemented constraints is performed. To evaluate the effectiveness of ISLa-generate precondition, several bugs are injected into the AIL processor, one at a time, and resulting buggy implementations are fuzz-tested with ISLa as a test input generator. When generating test inputs for each test case, ISLa constraints are introduced incrementally, measuring the number of test cases and time elapsed before producing a crash. This approach borrows ideas from mutation testing, in which the tester aims to find a set of test inputs which can be used to discriminate a correct program from programs "close" to the correct one [1]. However, the present study is limited to a small set of programs "close" to the correct one, i.e. one for each injected bug, which is used to evaluate the effectiveness of test input sets produced by ISLa to produce a crash, i.e. discriminate an almost correct program from the correct one.

ISLearn is claimed to infer constraints using the input grammar and an observable system [15]. Since each of the injected bugs leads to a crash of the program, a programmer crash could be used as a property mine condition that could produce input which triggers the erroneous behaviour.

The specific implementation will aim to follow the intended design but might deviate from it to work around the fundamental or practical limitations of ISLa and ISLearn. Just as the limitations themselves, these workarounds constitute useful insight and are reported in the present study to contribute to conclusions and inform suggestions for further work.

4 IMPLEMENTATION

To incorporate ISLa into the testing process more conveniently, a simple test framework in Python was developed (published under MIT Licence) [17]. It provides simple runtime observability of the system under test using stateful `Condition` objects, which are triggered in the code of tested system. Conditions can be composed and modified using boolean conjunction, disjunciton, and negation. While simple conditions can be triggered directly and are then considered *true*, composite conditions are evaluated as the value of the underlying boolean expression with values of constituent conditions substituted. Using these conditions, a tester can programmatically verify that a specific execution flow associated was triggered. The framework then provides a convenient way to decorate test functions, assigning them relevant conditions. In turn, the framework will create property functions to test whether a given input triggers a specific condition. Namely, the resulting property function accepts a test input; the test condition is reset, so that it can be triggered again by the system under test; then the test function is called on the test input; the test function is expected to pass the given input to the system under test, which in turn will trigger the conditions; afterwards the test condition is evaluated and it value is returned. There derived property functions are passed to ISLearn to mine preconditions and used in the experiments with injected bugs, to verify that specific buggy behaviours are triggered.

Throughout the case study, a few practical limitations or problems of ISLa were encountered, some fixed, others worked around. For instance, the ISLa solver would sometimes unpredictably get stuck generating new input samples. By default, ISLa generates one input each time the `ISLaSolver.solve()` method is invoked, with inputs typically getting longer and more complex with each subsequent one, and will raise a `StopIteration` exception if the solver claims that no more satisfying solutions can be found [16]. This seems to work well on the input languages evaluated in the study that introduces ISLa [15], however, when generating AIL with the grammar and constraints developed in this study, the solver would run out of solutions often before having generated 200 samples, while clearly more variations of inputs satisfying the constraints existed, e.g. obtained by re-initialising the solver and generating new inputs. In the present study, it has been worked around by generating inputs with `ISLaSolver.solve()` until `StopIteration` is raised, and then consequently mutating previously generated input with `ISLaSolver.mutate()`. When `ISLaSolver.mutate()`, eventually gets stuck, the solver is reinitialised and the previous procedure repeats. Note, that this is done on-demand, one input at a time, implemented at Python generator.

On another note, while ISLa uses text angle brackets to denote the input grammar's nonterminals, it provides no mechanism to escape them in case the input language in case the underlying language also uses these characters. This led to errors while parsing ISLa match expressions which contained these characters, which were common since AIL (like XML) uses angle brackets to denote tags. As a crude workaround, angle brackets in the AIL grammar were replaced by parentheses and then replaced back with angle brackets in the generated input sample. Similarly, double quotation marks were replaced by single quotation marks, to avoid unnecessary escaping.

Lastly, running some parts of ISLa resulted in unhandled runtime exceptions or logical errors. For example, calls to the built-in `count()` predicate consistently resulted in an `AttributeError`. In the `level()` predicate, there was a potential logical error, for which a solution was found and contributed back to ISLa [18].

5 RESULTS

5.1 Context-free conditions

All considered context-free constraints are, in principle, expressible in ISLa. With constraints CF1-CF6 (corresponding ISLa code is listed in Appendix A) implemented in ISLa, a benchmark was performed

measuring mean time, across 10 runs, to generate fixed numbers (1, 10, and 100) inputs. On the first iteration, ISLa is only provided with the CFG for XML, as defined in The fuzzing Book [24]; on the second iteration, a constraint for matching XML open and close tags [15] is added; subsequently, constraints which model conditions CF1-CF6 are added, one per iteration.

Table 1. Time to generate N inputs satisfying AIL's context-free conditions

ISLa Constraint	**Time (s.) to N inputs**		
	N=1	N=10	N=100
None	0.071	0.112	0.68
+balanced xml	0.125	1.745	6.771
+correct tags (CF1)	1.423	1.588	6.582
+correct attrs. (CF2)	1.522	1.523	6.408
+build tag (CF3)	1.868	2.693	21.839
+text only in step (CF4)	1.707	3.188	29.95
+step inside task (CF5)	>1800*	>1800*	>1800*
+dep inside task (CF6)	>1800*	>1800*	>1800*

* a benchmark is considered *timed-out* after 30 min. (1800 sec.)

Since test fuzzing entails generating a massive amount of test inputs, the time it takes to generate an input sample should be limited. In the present study, a limit of 30 minutes[1] is set, after which the attempt is considered to have *timed out*. This limit is quite far above what would be practical for test input generation, where, typically, multiple inputs are produced per minute. However, including data points up to 30 minutes in cases of inadequate performance allows to reason about the extent to which the performance is not adequate. As seen in Table 1, after adding the constraint for CF5, input generation times out even when attempting to obtain a single input.

5.2 Context-sensitive conditions

Listing 3. ISLa for CS1 - all IDs are unique

```
forall <task>="(task id='{<id> task_id}')<deps><
    steps>(/task)":
  (not (exists <task> other="(task id='{<id>
      other_task_id}')<deps><steps>(/task)":
    (task_id = other_task_id)))
```

Since the performance issues observed when expressing context-free conditions in ISLa would complicate the effective evaluation of the other, context-sensitive, conditions, the context-free conditions were instead encoded into a CFG (listed in Appendix A). With that, context-sensitive conditions CS1-CS3 could be directly described using only SMT expressions, boolean expressions and tree quantifiers in ISLa [15].

Listing 4. ISLa for CS2 - only use existing task IDs in dependencies

```
forall <dep>="(dep id='{<id> dep_id}'/)":
  exists <task>="(task id='{<id> task_id}')<deps
      ><steps>(/task)":
    (dep_id = task_id)
```

[1]benhcmarks are measured on a 2,6 GHz 6-Core Intel Core i7 CPU with 16 GB 2667 MHz DDR4 Memory running macOS 13.6.1

Table 2. Time to generate N inputs satisfying AIL's context-sensitive conditions

ISLa Constraint	**Time (s.) to N inputs**		
	N=1	N=10	N=100
None	0.407	0.928	2.158
+unique IDs (CS1)	0.759	0.612	13.451
+existing deps (CS2)	0.157	0.122	14.903
+one entry point (CS3)	1.176	1.105	33.908
+no self-deps (rel. CS4)	1.735	1.337	34.539
+no orphans tasks (rel. CF5)	1.288	1.251	37.091

For CS3, a potential simple implementation could be possible using the `count(tree, NEEDLE, NUM)` predicate built into ISLa, which constraints the number of occurrences of the `NEEDLE` nonterminal inside of the `tree` to `NUM`. However, since calling this predicate with `start`, i.e. the root of the derivation tree, as `tree` resulted in an unhandled runtime exception in ISLa, an alternative implementation of the condition using quantifiers was chosen instead.

Listing 5. ISLa for CS3 - exactly one main entry point

```
forall <main-true> a in start:
  forall <main-true> b in start:
    (same_position(a, b)))
and (exists <mb-main> main in start:
  (main = " main='true'")
```

Unlike CS1-CS-3, CS4 (no circular dependencies between tasks) could not be directly expressed in ISLa. Since tasks form a direct graph where edges represent dependencies, formally describing a cycle, i.e. a path of tasks $\langle t_0; t_1; \ldots; t_k \rangle$ where $t_0 = t_k$ and the path contains at least one edge, would require an expression binding an arbitrary number of variables, one for each task in the cycle. Since ISLa does not feature anything akin to cycles or recursion, each expression can only operate on nodes bound to variables either explicitly by match expressions or implicitly by using free-nonterminals. Therefore a constraint over an arbitrary number of nodes, such as the no-cycle constraint is not expressible in ISLa. However, a constraint expressing the presence or absence of a cycle of a certain fixed length is, in principle possible. For instance, the simplest cycle, where a task depends on itself could be expressed by the constraint in Listing 6.

Listing 6. ISLa for CS4 - no task depends on itself

```
forall <dep> d="(dep id='{<id> dep_id}'/)":
  exists <task> t="(task id='{<id> task_id}')<
      deps><steps>(/task)":
    ((dep_id = task_id) and (not (inside(d, t)))
      )
```

This means that, in principle, this constraint could be expressed for all cycles in a certain range of lengths, but it must be noted that a definition for each specific length will incur a separate performance cost.

Another condition that cannot be directly expressed in ISLa is CS5, which specifies that all tasks must be in the dependency tree of the main entry point. Much like in CS4, this would require considering a

path in the task dependency graph consisting of an arbitrary length of tasks, which is not possible in ISLa. However, it is not difficult to specify that each task, except for the main entry point is depended on by at least one task, as seen in Listing 7.

Listing 7. ISLa for CS5 - no orphaned tasks

```
1 forall <task>="(task id='{<id> task_id}')<deps><
      steps>(/task)":
2   exists <dep> d="(dep id='{<id> dep_id}'/)":
3     (dep_id = task_id)
```

Note, that the code in Listing 7 does not specify that a task is depended on by a *different* task, but rather simply that there is a dependency with the matching ID. However, in combination with the code in Listing 6, it guarantees the absence of *orphaned* tasks, i.e. tasks on which no other task depends. This constraint still does not guarantee that all tasks are in the dependency tree of the main entry point, but it excludes one of the way in which this constraint is violated and makes it more likely to hold, especially in small inputs containing few tasks.

While the ISLa solver still generally slows down upon adding each consecutive constraint, sample inputs for all implemented context-sensitive constraints were successfully generated in time shown it Table 2.

5.3 Evaluating effectiveness of test inputs

To evaluate the quality of test inputs, i.e. their effectiveness in finding erroneous behaviours in the system under test, multiple bugs were manually injected into different parts of the program that processes AIL input. As seen in Table 3, when using relevant ISLa constraints in combination with the AIL CFG, generated test inputs lead to triggering the crashes caused by the injected bug sooner.

Table 3. Evaluating the quality of test inputs generated with ISLa using relevant constraints

ISLa Constraint	**Inputs until crash**		
	Bug #1	**Bug #2**	**Bug #3**
None	78.2/2.220s	353.6/9.73s	*timeout**
+CS1	1.2/0.146s	47.2/4.29s	576.3/63.842s
+CS2	1.0/1.506s	55.4/24.084s	41.0/8.94s
+CS3	1.0/1.419s	48.6/18.484s	-**
+no self-dep.	1.0/1.887s	37.0/12.897s	1.4/3.008s***
+no orphaned tasks	1.0/1.846s	44.8/15.469s	-**

* if no crash produced in 30m or 1000 inputs, the benchmark is considered timed out

** skipped for the specific for performance reasons

*** inverted for the specific use case

5.4 Mining preconditions with ISLearn

A simple runtime observability system has been set up, using flags which are set by the system under test in select places in its execution flow, specifically, when the system encounters an error in the input and exists early, or when an exception cause by one of the injected bugs is raised. The value of a flag can be accessed after the program terminates, either successfully or with an error, to check whether the corresponding runtime behaviour was triggered. This allows using a combination of the programme and a flag to be used as a *property* function which checks whether a given test input trigger the program behaviour corresponding to the flag. Such a property function together with the input grammar technically satisfies the minimum requirements for ISLearn to mine invariants of inputs which satisfy the property, i.e. trigger the desired runtime behaviour of the programme [15]. However, when used to mine conditions for valid AIL input, ISLearn terminated with no results, failing to find positive examples for learning, which means that among the inputs generated based on the input grammar, none happened to trigger the desired runtime behaviour. To work around this issue, inputs were fuzzed with ISLa using additional constraints to increase the relative frequency of positive examples. Despite providing both positive and negative examples, no constraints were mined by ISLearn on AIL.

6 DISCUSSION

Overall, the effectiveness of ISLa in fuzz-testing AIL has varied between different sub-problems. When it comes to describing invariants of valid test inputs, ISLa is generally expressive enough to afford context-free constraints. However, even with the number of constraints in a simple input format like AIL, it is impractical to generate test inputs with ISLa due to its poor performance in the presence of multiple constraints. Performance is significantly improved by expressing the context-free invariants using CFG and restricting the use of ISLa constraints to describe context-sensitive invariants.

ISLa has been successfully used to express context-sensitive preconditions for specific test cases. Preconditions expressed in ISLa have been shown to increase the likelihood of reaching specific behaviours of the system under test which are relevant to the test case, therefore revealing the bugs in the system sooner compared to generating inputs based on the CFG alone. Still, while using ISLa constraints improved the quality of test inputs, not all preconditions could be exactly expressed in ISLa. Preconditions, which use an arbitrary number of nodes are impossible since all the nodes must be bound to separate variables. In some cases, language-level ISLa predicates such as `count()` and `inside()` can be used to describe constraints not possible or inconvenient otherwise. However, ISLa currently does not include predicates necessary to describe constraints over task dependencies in AIL, specifically the absence of cyclic dependencies and all tasks being contained in the dependency tree of the main entry point. Note, that it is possible, that in some cases where it is not possible to directly describe the desired constraint, an expressible equivalent might exist. However, in the present study, such equivalents for the task dependency constraints were not found, instead particular cases of invariants were implemented, which did not guarantee conforming inputs but increased the likelihood of generating one. Regardless, the ability to express conditions involving an arbitrary number of nodes is dependent on the existence of simpler equivalents or the presence of relevant language-level predicates.

When it comes to quality of the generated test input, using relevant ISLa constraints as test-case preconditions is shown to increase the efficiency of revealing bugs. As seen in Table 3, it decreased

the number of inputs and time needed to find Bug #1, which was already found with no too much difficulty when no constraints were used. Not only that, but it allowed to efficiently find Bug #3, which otherwise was not discovered within the time limit at all. However, in case of Bug #2 the improvements were less significant, and the addition of the last precondition led to a worse performance than with the combination of the previous four.

Note also, that for Bug #1, after adding the first two conditions, the crash was always triggered on the first try, however, adding each consequent constraint led to diminishing returns in terms of time required to generate the test input. Together with benchmarks listed in Table 1 and Table 2, this shows how the time to generate inputs noticeably increases when adding more constraints.

6.1 Threats to validity

While this case study attempts to draw generalised conclusions about the usability of ISLa for fuzz-testing, the scope of it is narrow. Only one case study is considered, namely AIL, which is designed to resemble real-world build systems like GNU Make and Maven but is nevertheless a simplified example of such a system. The performance of ISLa on a larger system, with potentially more complex input constraints and a larger code base may differ.

Additionally, ISLa and more so ISLearn seem to be in a fairly unstable phase of development, with some instances of crashing and failing in unpredictable ways despite following the documentation. As the result, several experimental setups were rejected simply due to producing an unexpected crashes. Additionally, it could suggest that some performance limitations faced might be a consequence of a bug in the implementation.

7 RELATED WORK

7.1 Fuzz-testing

Test input fuzzing has seen a lot of contributions improving its effectiveness in discovering software bugs and vulnerabilities. In particular, coverage-guided approaches have proven effective [21, 23]. Some systems, like interpreters and compilers, define complex requirements for valid inputs, to which end grammar-based fuzzing techniques have been used to generate inputs which conform to the target grammar [8, 9]. Grammar-aware and coverage-based approaches have also been combined in grammar-aware grey-box fuzzing to surpass the out-of-the-box performance of AFL [19]. Alternatively, evolutionary algorithms that select for test inputs which trigger certain observable behaviours, e.g. runtime exceptions, can also be used to fine tune fuzzers for better coverage and more effective discovery of erroneous behaviours [3].

While all the aforementioned approaches aim to be generic, some more complex input validity requirements are better handled with specialised tools. For instance, CSmith, a tool for generating C code for compiler testing, in addition to generating structurally and semantically valid C code, also guarantees that the generated programs do not contain undefined behaviour [22]. Polyglot tools that can handle such complex semantics, e.g. XSmith, also exist, however, they require a complex input specification [7].

7.2 Property-based testing

Similarly to grammar-based fuzzing, property-based testing (PBT) aims to comprehensively test programs by generating large amounts of random test input [5]. Unlike the former, it relies on a more extensive program behaviour specification consisting of executable *properties*, often in the form of preconditions, satisfied by input, and postconditions, which should be satisfied by the output of the programme. Such a specification is a significantly more complete description of the intended program behaviour than just an input grammar, which comes at a cost of more demanding implementation and verification.

While PBT often avoids the problem of generating structurally valid inputs by relying on data structures of the host language, efficiently generating precondition-satisfying input is often challenging [5, 11, 12]. In the quest for an efficient and straightforward mechanism for generating precondition-satisfying test inputs a range of approaches has been explored, including domain-specific languages for defining input generators [10], search-based test generation strategies to more efficiently explore the input space [12, 14], and modelling generators as parsers of random streams of decisions [6]. Ideas from coverage-guided fuzzing have also been successfully incorporated into PBT, specifically, for identifying, retaining, and mutating those test inputs which expand control-flow coverage, therefore, increasing the probability of finding precondition-satisfying inputs [11]. It is claimed to perform significantly better than random testing in cases that require satisfying space preconditions [11].

8 CONCLUSION

Relying on the observations and results of the AIL case study, we draw the following answers to the research questions.

> **RQ1** *How well can testing preconditions be expressed in ISLa?*

ISLa is decently effective in expressing test preconditions. As shown in the case study on AIL, ISLa as a specification language is suitable to express conditions relevant to test input. However, ISLa generally does not afford expressing conditions, whose computational description requires an unbounded number of derivation tree nodes. Note, that some specific cases of this limitation are addressed by supporting language-level predicates which allow expressing some conditions not expressible otherwise, which is only a partial solution to the problem, as it leaves the user dependent on the availability of relevant predicates. On the practical side, the performance of the ISLa solver when combining several conditions is inadequate for use in fuzz-testing.

> **RQ2** *How effective is ISLa at generating test inputs?*

The technique of generating test inputs with ISLa using test-relevant input preconditions has been shown to improve the efficiency of revealing bugs in the system under test compared to generating test input based on the input CFG alone.

> **RQ3** *What preconditions can be mined with ISLearn?*

In response to **RQ3**, we conclude that ISLearn is, in its current state, not a suitable tool to infer test-relevant preconditions by observing the system under test.

In summary, ISLa has been shown to effectively tackle some of the challenges in fuzz-testing programs which accept an input structured by a CFG. However, this tool has significant limitations in the types of test conditions which can be expressed and its runtime performance when generating test inputs. For instance, it is unlikely to be suited to fuzz compilers with complex static analysis, both due to its limitations in expressiveness and risk of poor performance as a result of a high number of constraints needed to describe the semantics of the input programming language. In these instances, it is ideal to use specialised tools, e.g. CSmith [22], or polyglot tools such as XSmith [7]. Nevertheless, in cases of test input languages with relatively simple semantic invariants, the proposed testing setup based on ISLa is a viable, potentially less labour-intensive alternative to grammar-based fuzzers such as Grammarinator [8], allowing declarative specification of both the input language and the test-case preconditions.

9 FUTURE WORK

Performance of ISLa, specifically measured as the time it takes for the solver to generate a fixed large number of inputs, has been unpredictable over the course of developing, debugging, and benchmarking ISLa constraints presented in this study. For instance, when constraints were added incrementally, the performance on the final iteration, with all constraints in conjunction varied greatly depending on the order in which the constraints were introduced, despite the equivalence of final combined constraints. More research could be done on how reordering constraints impacts the performance, and potentially how ISLa constraints can be automatically transformed into equivalent constraints with better runtime performance.

ISLa's model of input invariants based on quantifiers and boolean expressions makes for simple semantics and a simple declarative syntax. However, it has inherent limitations, such as the inability to describe constraints over an arbitrary number of nodes (in Section3.2, CS4 and CS5). In ISLa, this problem is already addressed by including predicates such as `count()` and `same_position()`. This study shows that ISLa is currently incapable of expressing constraints involving quantifying an arbitrary number of nodes outside of what is allowed by built-in predicates. Further research could attempt to formalise the limits of what invariants can be described by ISLa and find ways to expand those limits, for instance, by implementing additional predicates.

REFERENCES

[1] Budd, T. A., DeMillo, R. A., Lipton, R. J., and Sayward, F. G. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1980), POPL '80, Association for Computing Machinery, p. 220–233.

[2] Docker, Inc. Compose specification - services. https://docs.docker.com/compose/compose-file/05-services/, 2024. Accessed: 2024-06-22.

[3] Eberlein, M., Noller, Y., Vogel, T., and Grunske, L. Evolutionary grammar-based fuzzing. In *Search-Based Software Engineering* (Cham, 2020), A. Aleti and A. Panichella, Eds., Springer International Publishing, pp. 105–120.

[4] Free Software Foundation, Inc. *GNU Make: A Program for Directing Recompilation*, 2023. Accessed: 2024-06-22.

[5] Goldstein, H., Cutler, J. W., Dickstein, D., Pierce, B. C., and Head, A. Property-based testing in practice. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (New York, NY, USA, 2024), ICSE '24, Association for Computing Machinery.

[6] Goldstein, H., and Pierce, B. C. Parsing randomness. *Proc. ACM Program. Lang. 6*, OOPSLA2 (oct 2022).

[7] Hatch, W., Darragh, P., Porncharoenwase, S., Watson, G., and Eide, E. Generating conforming programs with xsmith. In *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (New York, NY, USA, 2023), GPCE 2023, Association for Computing Machinery, p. 86–99.

[8] Hodován, R., Kiss, A., and Gyimóthy, T. Grammarinator: a grammar-based open source fuzzer. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation* (New York, NY, USA, 2018), A-TEST 2018, Association for Computing Machinery, p. 45–48.

[9] Holler, C., Herzig, K., and Zeller, A. Fuzzing with code fragments. In *21st USENIX Security Symposium (USENIX Security 12)* (Bellevue, WA, Aug. 2012), USENIX Association, pp. 445–458.

[10] Lampropoulos, L., Gallois-Wong, D., Hriţcu, C., Hughes, J., Pierce, B. C., and Xia, L.-y. Beginner's luck: a language for property-based generators. *SIGPLAN Not. 52*, 1 (jan 2017), 114–129.

[11] Lampropoulos, L., Hicks, M., and Pierce, B. C. Coverage guided, property based testing. *Proc. ACM Program. Lang. 3*, OOPSLA (oct 2019).

[12] Löscher, A., and Sagonas, K. Targeted property-based testing. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (New York, NY, USA, 2017), ISSTA 2017, Association for Computing Machinery, p. 46–56.

[13] Mernik, M., Črepinvsek, M., Kosar, T., and Žumer, D. Grammar-based systems: Definition and examples. *Informatica 28* (11 2004), 245–255.

[14] Olsthoorn, M., van Deursen, A., and Panichella, A. Generating highly-structured input data by combining search-based testing and grammar-based fuzzing. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (New York, NY, USA, 2021), ASE '20, Association for Computing Machinery, p. 1224–1228.

[15] Steinhöfel, D., and Zeller, A. Input invariants. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (New York, NY, USA, 2022), ESEC/FSE 2022, Association for Computing Machinery, p. 583–594.

[16] Steinhöfel, Dominic. Isla specification. https://isla.readthedocs.io/en/latest/islaspec.html, 2023. Accessed: 2024-06-22.

[17] Vasylenko, M. Fuzz-testing with isla. https://github.com/vasylenson/string-theory-py, 2023. GitHub repository.

[18] Vasylenko, M. Isla repository, github. pull request #94: Fix level predicate. https://github.com/rindPHI/isla/pull/94, June 2024. Pull request #94, Isla Repository, GitHub.

[19] Wang, J., Chen, B., Wei, L., and Liu, Y. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (2019), pp. 724–735.

[20] World Wide Web Consortium (W3C). HTML5: A vocabulary and associated APIs for HTML and XHTML. https://www.w3.org/TR/html5/, October 2014. [Online; accessed 22-June-2024].

[21] Yan, S., Wu, C., Li, H., Shao, W., and Jia, C. Pathafl: Path-coverage assisted fuzzing. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security* (New York, NY, USA, 2020), ASIA CCS '20, Association for Computing Machinery, p. 598–609.

[22] Yang, X., Chen, Y., Eide, E., and Regehr, J. Finding and understanding bugs in c compilers. *SIGPLAN Not. 46*, 6 (jun 2011), 283–294.

[23] Zalewski, M. Americal fuzzy lop (fuzzer). https://lcamtuf.coredump.cx/afl/technical_details.txt.

[24] Zeller, A., Gopinath, R., Böhme, M., Fraser, G., and Holler, C. Index. In *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2020. Retrieved 2020-09-27 19:14:05+02:00.

APPENDIX A

ISLa formula expressing the context-free constraints of AIL.

```
# matching open and close XML tags
(forall <xml-tree> tree="<{<id> opid}[ <xml-attribute>]><inner-xml-tree></{<id> clid}>" in start:
    (= opid clid))
and
# correct tag IDs
(<xml-open-tag>.<id> = "build" or <xml-open-tag>.<id> = "task" or <xml-open-tag>.<id> = "step") and (<xml-openclose-tag>.<id> = "dep")
and
# correct attribute IDs
<xml-attribute>.<id> = "id" or <xml-attribute>.<id> = "main" or <xml-attribute>.<id> = "cost" or <xml-attribute>.<id> = "script"
and
# the build tag at the top level
(exists <xml-tree> root="<{<id> id}>{<inner-xml-tree> inside}<xml-close-tag>" in start: (id = "build"))
and
(forall <xml-tree> root="<{<id> id}[ <xml-attribute>]><inner-xml-tree><xml-close-tag>" in start: (
    (id = "build") implies direct_child(root, start)))
and
# only steps may contain text
(forall <xml-tree> command="<{<id> id}><text><xml-close-tag>" in start: (id = "step"))
and
# <step> is always inside <task>
(forall <xml-tree> step="<{<id> step_id}[ <xml-attribute>]><inner-xml-tree><xml-close-tag>" in start: (
    (step_id = "step") implies
    (exists <xml-tree> task="<{<id> task_id}[ <xml-attribute>]><inner-xml-tree><xml-close-tag>":
        (task_id = "task" and inside(step, task)))))
and
# <dep> is always inside <task>
(forall <xml-tree> dep="<{<id> dep_id}[ <xml-attribute>]/>" in start: (
    (dep_id = "dep") implies
    (exists <xml-tree> task="<{<id> task_id}[ <xml-attribute>]><inner-xml-tree><xml-close-tag>":
        (task_id = "task" and inside(dep, task)))))
```

APPENDIX B

Final Contex-Free Grammar for AIL (in EBNF notation).

```
<start> ::= <build>
<build> ::= "(build)" <tasks> "(/build)"
<tasks> ::= <task> | <task> <tasks>
<task> ::= "(task id='" <id> "'" <mb-main> ")" <deps> <steps> "(/task)"
<mb-main> ::= "" | <main-true>
<main-true> ::= " main='true'"
<deps> ::= "" | <dep> <deps>
<dep> ::= "(dep id='" <id> "'/)"
<steps> ::= <step> | <step> <steps>
<step> ::= "(step " <cost> ")" <command> "(/step)" | "(step " <cost> " " <script> ">" "(/step)"
<cost> ::= "cost='" <int> "'"
<script> ::= "script='" <text> "'"
<id> ::= <text>
<command> ::= <text>
<text> ::= <char> | <char> <text>
<char> ::= <letter> | <digit> | <special>
<int> ::= <leaddigit> | <leaddigit><digits>
<digits> ::= <digit> | <digit> <digits>
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
<leaddigit> ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
<letter> ::= "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" | "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z"

<special> ::= "_" | "-" | "!" | "@" | "#" | "$" | "%" | "^" | "*" | "+" | "=" | "|" | ":" | " " | ";" | "," | "." | "?" | "/"
```

Due to apparent lack of escaping in ISLa, the grammar does not describe a subset of XML. Angle brackets <> are substituted by round brackets () and double quotes " are replaced by single quotes '. Once a sentence was produced by ISLa it would be converted to an XML compatible string using Python's `str.replace()`:

```
def br_to_xml(inp: str):
    for old, new in [("'", '"'), ('(', '<'), (')', '>')]:
        inp = inp.replace(old, new)
    return inp
    return inp
```