Fuzzing Class Specifications

Facundo Molina
University of Rio Cuarto and CONICET
Argentina
fmolina@dc.exa.unrc.edu.ar

Marcelo d'Amorim
Federal University of Pernambuco
Brazil
damorim@cin.ufpe.br

Nazareno Aguirre
University of Rio Cuarto and CONICET
Argentina
naguirre@dc.exa.unrc.edu.ar

ABSTRACT

Expressing class specifications via executable constraints is important for various software engineering tasks such as test generation, bug finding and automated debugging, but developers rarely write them. Techniques that infer specifications from code exist to fill this gap, but they are designed to support specific kinds of assertions and are difficult to adapt to support different assertion languages, e.g., to add support for quantification, or additional comparison operators, such as membership or containment.

To address the above issue, we present SpecFuzzer, a novel technique that combines grammar-based fuzzing, dynamic invariant detection, and mutation analysis, to automatically produce class specifications. SpecFuzzer uses: *(i)* a fuzzer as a generator of candidate assertions derived from a grammar that is automatically obtained from the class definition; *(ii)* a dynamic invariant detector –Daikon– to filter out assertions invalidated by a test suite; and *(iii)* a mutation-based mechanism to cluster and rank assertions, so that similar constraints are grouped and then the stronger prioritized. Grammar-based fuzzing enables SpecFuzzer to be straightforwardly adapted to support different specification languages, by manipulating the fuzzing grammar, e.g., to include additional operators.

We evaluate our technique on a benchmark of 43 Java methods employed in the evaluation of the state-of-the-art techniques GAssert and EvoSpex. Our results show that SpecFuzzer can easily support a more expressive assertion language, over which is more effective than GAssert and EvoSpex in inferring specifications, according to standard performance metrics.

CCS CONCEPTS

• **Theory of Computation → Program specifications**; • **Software and its engineering → Dynamic analysis**; **Software testing and debugging**.

KEYWORDS

Oracle problem, specification inference, grammar-based fuzzing.

ICSE 2022, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/1122445.1122456

ACM Reference Format:
Facundo Molina, Marcelo d'Amorim, and Nazareno Aguirre. 2022. Fuzzing Class Specifications. In *Proceedings of The 44th International Conference on Software Engineering (ICSE 2022).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Software specifications are abstract descriptions of the software's intended behavior. They serve two main purposes: to explicitly state the user needs and to check implementation conformance [23]. In Object-Oriented (OO) design, where software is organized as a set of classes, a class specification describes the intended behavior of the class methods and the constraints on the state of the class' objects. While the specification of a class is typically described *informally*, through natural language documentation of its API, the specification becomes significantly more useful when expressed *formally*, through constraints known as *contracts*[1] [36, 46]. Contracts enable techniques of various kinds, including test generation [6, 13, 33, 48], automated debugging [16, 34, 43, 44], bug finding [30, 41], and verification [17, 21, 22, 30]. Despite the benefits of formal contracts, developers rarely write them.

To aid developers in equipping implementations with contracts, techniques for inferring class specifications have been proposed [9, 18, 38, 47]. However, the specification expressiveness of these approaches is limited. Daikon [18], the baseline that other techniques use, supports a restricted set of templates, from which assertions are generated. It is then limited to simple assertions (e.g., no direct support for quantification), or requires the developer to manually extend the assertion language. GAssert [47] and EvoSpex [38], two recently proposed techniques for contract inference, try to address this limitation of Daikon by supporting more expressive assertion languages, but their extensions focus on specific kinds of constraints: GAssert focuses on logical/arithmetic constraints (no quantified expressions) and EvoSpex focuses on object navigation constraints (only very simple logical and arithmetic operators are supported). Moreover, as both techniques are based on evolutionary search, they are difficult to extend or adapt to support further expressions, as the evolutionary algorithms are targeted for the specific languages supported by the corresponding tools.

To overcome the limitations of existing approaches, we propose SpecFuzzer, a technique for generating likely specifications by *fuzzing* potential specifications associated with a given class. SpecFuzzer uses grammar-based fuzzing to automatically generate constraints that can be used as candidate specifications by an invariant detection tool (in our case, we use Daikon). Fuzzing [50],

[1] In the context of this paper, we will interchangeably use the terms *contract* and *specification*. A contract is typically composed of different assertions for various program points, such as method preconditions and postconditions.

traditionally used to efficiently produce structured random data for testing, has two key advantages in this context: (1) it eliminates the need of developers to manually define candidate assertions and (2) it enables developers to straightforwardly adapt the language of assertions by manipulating the fuzzing grammar.

Fuzzing can quickly produce very large sets of assertions to be fed to a dynamic detection tool. However, as the assertions are generated randomly, and dynamic invariant detection only filters out assertions that can be invalidated by a given test suite, a substantial number of candidate specification expressions may be reported by the dynamic detector, when fed with fuzzed assertions. To address this problem and better assist developers in driving their attention to the likely most relevant specifications, SpecFuzzer uses an assertion reduction mechanism based on clustering and mutation testing [7, 42]. After generating thousands of candidate specifications with fuzzing, SpecFuzzer uses the output of a custom mutation analysis to cluster candidate specifications. More precisely, it partitions the set of specifications according to the mutants they kill, and within each partition, the assertion that is falsified the most number of times when running the test suite on the mutants, is picked as the representative. Notice that, even though all the assertions in the same partition kill the same mutants, some may be falsified more than others, as a same mutant may be killed by multiple tests. The rationale for the mutation based partition is that assertions that kill different mutants are non-equivalent (or, alternatively, that assertions that kill the same mutants are "similar"); the rationale for ranking assertions according to the number of failures is that assertions that are falsified a greater number of times are "stronger".

We compared SpecFuzzer with GAssert [47] and EvoSpex [38], two state-of-the-art techniques in specification inference. To evaluate SpecFuzzer, we used the same benchmarks from the evaluation of GAssert and EvoSpex, carefully studied the subjects, and manually produced corresponding "ground truth" assertions capturing the intended behavior of the subjects. We then used this ground truth to accurately assess precision and recall of SpecFuzzer, GAssert, and EvoSpex. It is worth noting that (1) prior work used indirect metrics to compute precision and recall (as opposed to the direct usage of ground truth) and (2) prior work used subsets of the subjects we consider (our benchmark is the combination of the GAssert and EvoSpex benchmarks). Our results show that SpecFuzzer increases the expressiveness over GAssert and EvoSpex, being able to express ~45% more assertions in the ground truth than these tools. SpecFuzzer was also able to detect 75% of all assertions in the ground truth, showing a better overall performance compared to previous techniques. The results we obtained provide initial, yet strong evidence that SpecFuzzer is effective.

In summary, this paper makes the following contributions:

- SpecFuzzer, a novel technique for assertion inference that combines grammar-based fuzzing and dynamic invariant detection.
- An efficient mechanism for grouping similar assertions and for ranking assertions based on their strength.
- A thorough evaluation of our technique against GAssert and EvoSpex, in which performance metrics are computed in relation to manually written assertions (ground truth).

The evaluation artifacts of SpecFuzzer are publicly available [5].

2 BACKGROUND

This section presents background material that is important for the rest of the paper.

2.1 Specification Inference

Specification inference is the problem of generating a formal description of the software behavior from existing software artifacts, e.g., documentation, source code, etc. Specification inference is closely related to the *oracle problem* [8], which is the problem of deciding whether or not a program execution is consistent with the desired behavior of the program. Specification inference provides a means to create *oracles* [8]. For regression testing purposes, it sometimes suffices to produce specifications of expected properties as assertions for the context of a given test case [20]. However, more general assertions that capture properties *at given locations within the program (not the test) for any input* have other applications, including testing. This is the problem we study in this paper, defined as follows.

Definition 2.1. Given a target program $\mathcal{P}$, and a program point of interest ρ in $\mathcal{P}$, infer a specification ϕ that captures the states at ρ, i.e., for every state s of $\mathcal{P}$, ϕ holds in s if and only if there exists an execution t of $\mathcal{P}$ such that s is the state of t at program point ρ.

2.2 Grammar-based Fuzzing

Fuzzing is a very active topic both in research [1] and practice [2–4]. Fuzzing is a technique to automatically produce large sets of (often structured) data, for testing a target program. The generation process typically involves randomness and the rationale is that testing on (large sets of) quasi-valid data can reveal subtle bugs, such as wrongly handled inputs and corner cases. A well-known use case of fuzzing is detection of security vulnerabilities, such as buffer overflows [35, 37].

Different fuzzing strategies exist [35]. Grammar-based fuzzing uses an input grammar to produce syntactically-valid inputs by traversing the production rules of the grammar. In its simplest form, the input generation process can be implemented as an incremental expansion of a string starting from the initial grammar symbol, and replacing non-terminal symbols by the application of a randomly-chosen production rule of the corresponding non-terminals, until the string consists of terminals only; a bound on the number of non-terminals enables this process to handle recursion, which would otherwise lead to infinite loops. As an example, consider a scenario where the program to test takes as input a propositional logic (PL) formula, characterized by the grammar from Figure 1. To generate testing data, the PL grammar can be fed to a grammar-based fuzzer (e.g., Grammarinator [25]) to efficiently obtain a very large set of well-formed test data (PL formulas, in this case). To generate inputs, the fuzzer explores paths induced by the grammar production rules. For instance, the input `neg(p and q)` can be obtained through the following derivation: *start* $\rightsquigarrow$ *formula* $\rightsquigarrow$ neg *formula* $\rightsquigarrow$ neg (*formula* and *formula*) $\rightsquigarrow$ neg (p and *formula*) $\rightsquigarrow$ neg (p and q). It is worth noticing that a great advantage of fuzzing in this case is that the input language can be easily adapted by modifying the grammar. For instance, our fuzzer would be able to generate

⟨*start*⟩ ::= ⟨*formula*⟩
⟨*formula*⟩ ::= ⟨*atomic*⟩ | neg ⟨*formula*⟩ | (⟨*formula*⟩ and ⟨*formula*⟩)
⟨*atomic*⟩ ::= true | false | p | q | r | ...

Figure 1: Propositional Logic grammar.

formulas with disjunctions if we add a corresponding production rule to the non-terminal *formula*.

The above example is relevant because the technique we propose in this paper uses grammar-based fuzzing as a lightweight approach to produce assertions (such as the PL formulas above) as candidate specifications for program points. The simplicity with which the grammar can be adapted or extended will be one of the advantages of the approach, compared with related techniques.

2.3 Assertion Language

An assertion is a logical expression associated with a program point expressing an expected property at that location. The use of assertions has wide-spread applications in software design [36], software testing [7], and verification [12, 24].

Our assertion language is similar in expressive power to JML, and features first-order quantification (`\forall`, `\exists`), arithmetic and logical operators, and reachability expressions (the reach operator `\reach(x, f1, ..., fk)` denotes the smallest set of objects reachable from `x`, through fields `f1`, ..., `fk`). Additionally, postcondition assertions might use the `\old(expr)` notation, to refer to the value of expression `expr` at the precondition. For simplicity, we drop the backslashes, shorten the quantifier names, and replace the semicolon notation in JML quantification by the implication (in the case of universal quantification) or conjunction (in the case of existential quantification). As an example, the following expression

```
all SortedList l:
    reach(this, next).has(l) ==> l.elem == old(l.elem)
```

states that the integer field `elem` of the list nodes reachable from `this` (`has` is the JML operator for membership) remains unchanged. This expression corresponds to the following JML expression:

```
\forall SortedList l;
    \reach(this, next).has(l); l.elem == \old(l.elem)
```

Our assertion language is motivated by the expressive power of the languages in related work, and in contract languages [11, 18, 19, 36, 38, 47]. This is a *general* language, that includes the usual relational, arithmetic and logical operators, but no domain specific functions (e.g., trigonometric functions, that would be relevant only for some analysis subjects, are not considered). The assertion language enables one to refer to class/object fields, but not to the results of method calls. That would require us to declare methods as "pure" to use them in assertions, which is beyond what our current implementation supports.

3 ILLUSTRATIVE EXAMPLES

This section illustrates SpecFuzzer on two simple examples with the purpose of (1) highlighting limitations of state-of-the-art specification inference techniques and (2) illustrate SpecFuzzer.

Examples. Figure 2 shows `min`, a Java method to compute the minimum of two integers, whereas Figure 3 shows `SortedList`, a Java

```
/* Returns the minimum of two integers */
public static int min(int x, int y) {
  if (x <= y) return x;
  else return y;
}
```

Figure 2: Method to get the minimum of two values.

```
public class SortedList {
  private int elem;
  private SortedList next;
  private static final int SENTINEL = Integer.MAX_VALUE;

  /* Constructors */
  public SortedList() { this(SENTINEL, null); }
  private SortedList(int elem, SortedList next) {
    this.elem = elem;
    this.next = next;
  }

  /* Method to insert an element in the list */
  void insert(int data) {
    if (data > elem) {
      next.insert(data);
    } else {
      next = new SortedList(elem, next);
      elem = data;
    }
  }
}
```

Figure 3: Class `SortedList` implements an ordered list of integers.

class implementing an ordered list of integers. The `min` method is straightforward. Class `SortedList` is slightly more elaborate. It has two instance fields, `elem` and `next`, that represent the value of a linked list node and the reference to the next node, respectively. It also has a class field (`SENTINEL`) that stores a special value –the maximum Java integer value– as a mark for the end of the list. The sentinel should be placed at the end of the list and should not be repeated. The default constructor creates a node marking the end of the list. The `insert` method takes the integer `data` as parameter and inserts it in its correct sorted position in the linked list. As it is not possible for any integer value to be greater than the sentinel, the search is guaranteed to insert the element before the sentinel.

Relevant Properties. The intended behavior of method `min` is that it computes the minimum between `x` and `y`. A specification of the postcondition of `min` in our assertion language is as follows:

```
(result == x || result == y) &&
(result <= x) && (result <= y)
```

The postcondition of method `SortedList.insert` involves various properties: the list is acyclic and sorted increasingly, the sentinel is in the list (at the end), and the `data` element is inserted. This postcondition can be specified as follows:

```
all SortedList l:
 reach(this, next).has(l) ==> !reach(l.next, next).has(l)
all SortedList l:
 reach(this, next).has(l) ==> l.elem <= l.next.elem
```

Table 1: GAssert and EvoSpex on the running examples.

	GAssert	EvoSpex
	min(int x,int y) - postcondition	
1	`(x > result && y == result) || (result <= y && result == x)`	`result <= x`
2		`result <= y`
	SortedList.insert(int data) - postcondition	
1	`elem-(data-elem) <= old(elem)`	`exists SortedList l: reach(this, next).has(l) && l.elem == data`
2		`old(elem) <= next.elem`

```
exists SortedList l:
 reach(this, next).has(l) && l.elem == SENTINEL
exists SortedList l:
 reach(this, next).has(l) && l.elem == data
```

We may consider these assertions to be the *ground truth* postcondition specifications of the corresponding methods, and what we would ideally expect specification inference tools to produce.

3.1 Techniques for Specification Inference

Daikon. Daikon [18] is dynamic technique that infers specifications by monitoring test executions. Considering Definition 2.1, besides the program $\mathcal{P}$, Daikon requires a test suite $\mathcal{T}$ for $\mathcal{P}$ to infer specifications. Daikon uses $\mathcal{T}$ to exercise $\mathcal{P}$; it monitors program states at various program points of $\mathcal{P}$; it considers a set of assertions obtained by instantiating assertion patterns, and those that are not invalidated by any test at a given program point are reported to the user as *likely invariants* at the program point.

GAssert and EvoSpex. GAssert [47] and EvoSpex [38] are recently proposed specification inference techniques. As Daikon, these tools execute a test suite of the program under analysis and observe executions to *infer* specifications that are consistent with the observations. While Daikon requires the test suite to be provided, GAssert and EvoSpex use their own test generation mechanisms (third-party test generation tools in the case of GAssert, a custom test generation approach in the case of EvoSpex). Although both techniques are based on evolutionary search, they have key differences. GAssert implements a co-evolutionary algorithm that explores the space of possible assertions (the co-evolution deals with false-positives and false-negatives via two cooperating evolutionary processes) and uses the OASIs [26] oracle assessment tool to iteratively improve the assertions. EvoSpex implements a classical genetic algorithm to explore the search space, and uses a state mutation technique to generate postcondition states in which the assertions being sought for should fail. GAssert's evolutionary operations focus on logical and arithmetic assertions whereas EvoSpex's focuses on object navigational properties. For these tools, changing the assertion languages implies redefining the corresponding evolutionary operators and other parameters of the evolutionary algorithms, which is non-trivial.

Tables 1 and 2 show how Daikon, GAssert, and EvoSpex perform on the examples (for brevity, we have removed `this` from non-quantified expressions in the `insert` example). GAssert performs perfectly on the `min` example, but poorly on `SortedList` (it does not capture most of the ground truth); EvoSpex infers one complex assertion for `SortedList.insert` (that the element is inserted) and misses the remaining three in the corresponding ground truth; it also infers part of the ground truth for `min`. Daikon infers the same as EvoSpex in the case of `min`, and in the case of `SortedList.insert`, it only infers specific sortedness instances between the first few elements of the list, but it fails to generalize this relationship for the whole structure. It fully misses the remaining assertions in the ground truth.

3.2 SpecFuzzer

SpecFuzzer uses a combination of static analysis, grammar-based fuzzing, and mutation analysis to infer specifications. SpecFuzzer proceeds as follows. First, it uses a lightweight static analysis to produce a grammar for the specification language, which is tuned to the software under analysis. Then, it uses a grammar-based fuzzer to generate candidate specifications from that grammar. A dynamic detector then determines which of those specifications are consistent with the behavior exhibited by a provided test suite. Finally, SpecFuzzer eliminates irrelevant and equivalent specifications using a mechanism based on mutation analysis and clustering. A salient feature of SpecFuzzer is that developers can adjust the set of specifications produced by tuning the grammar as opposed to making changes in the tool.

Table 2 shows the assertions that SpecFuzzer infers as postconditions for methods `min` and `SortedList.insert`. Recall that SpecFuzzer uses fuzzing and reports a higher number of assertions compared to the other techniques. We configured the fuzzer to produce 2000 candidate assertions per subject, and found that, out of those generated, 51 and 437 were confirmed as likely invariants by the dynamic detector for `min` and `insert`, respectively. The mutation-based partition strategy enabled SpecFuzzer to considerably reduce the reported assertions to 9 and 16, respectively.

In the case of `min`, the 9 inferred assertions are valid and their conjunction is equivalent to the corresponding ground truth. For `SortedList.insert`, the first 3 assertions already cover 3 out of 4 assertions in the ground truth (the only missing one is list acyclicity). The other inferred assertions are either valid but less relevant (4-13), or invalid (14-16). The invalid ones are specifications that were true in the provided test suite, but there exist some unseen scenarios in which they are falsified. Notice that this also affects the other techniques, even though GAssert and EvoSpex include costly mechanisms to reduce invalid assertions (the assertion inferred by GAssert for `SortedList.insert`, in particular, is an invalid property).

4 APPROACH

This section presents SpecFuzzer, a technique for specification inference that uses a combination of static analysis, grammar-based fuzzing, and mutation analysis.

Table 2: Daikon and SpecFuzzer on the running examples.

	Daikon	SpecFuzzer
	min(int x,int y) - postcondition	
1	result <= x	x >= result
2	result <= y	x < result ==> result <= 1
3		x >= y ==> y == result
4		x >= y || y != result
5		x <= y || y <= result
6		x <= y || y >= result
7		x >= y <==> y == result
8		x == y ==> y <= result
9		x <= y <==> x == result
	SortedList.insert(int data) - postcondition	
1	elem <= next.elem	exists SortedList l: reach(this, next).has(l) && l.elem == SENTINEL
2	elem <= next.next.elem	all SortedList l: reach(this, next).has(l) ==> l.elem <= l.next.elem
3	next.elem <= next.next.elem	exists SortedList l: reach(this, next).has(l) && l.elem == data
4	next != null	next != null
5	elem <= old(elem)	elem != old(elem) + 1
6	elem <= old(next.elem)	next.elem >= old(elem)
7	elem <= old(next.next.elem)	data >= next.elem || next.elem = old(elem)
8	elem <= data	elem == data xor data > old(elem)
9	next.elem >= old(elem)	elem > data ==> data = next.elem
10	next.elem <= old(next.elem)	exists SortedList l: reach(this.next, next).has(l) && l.elem != 1
11	next.elem <= old(next.next.elem)	exists SortedList l: reach(this, next).has(l) && l.elem > this.elem
12	next.next.elem >= old(elem)	exists SortedList l: reach(this, next).has(l) && l.elem <= l.next.elem
13	next.next.elem >= old(next.elem)	all SortedList l: reach(this.next, next).has(l) ==> l.elem >= this.next.elem
14	next.next.elem <= old(next.next.elem)	elem <= old(elem) ==> old(elem) < old(next.next.elem)
15		elem != next.next.elem + old(next.next.elem)
16		data >= next.next.elem || next.next.elem == old(next.elem)

Figure 4 shows the workflow of the technique. SpecFuzzer takes as input a Java[2] class C, and produces assertions that seek to characterize properties of different execution points in C, such as method preconditions and postconditions. Following the Daikon terminology, we will generally refer to assertions that hold on specific program points as *invariants*. The technique is organized as a pipeline of five components: (1) a *Tests and Mutants Generation* component that produces tests and mutants for other components of the pipeline, (2) a *Grammar Extractor* that analyzes C to generate a specification grammar for that class, (3) a *Grammar Fuzzer*, which produces candidate assertions by exploring the production rules from the extracted grammar, (4) a *Dynamic Invariant Detector*, responsible for inferring likely invariants from the fuzzed assertions via observations made with the executions of an input test suite, and (5) an *Invariant Selector* component, which partitions the likely invariants produced by the previous component to discard useless (weak) assertions, groups together similar assertions, and reports a reduced set of assertions, prioritizing the stronger ones. The following sections discuss these components in greater detail.

4.1 Tests and Mutants Generation

The first step of our process to infer specifications for a class C consists of *(i)* generating a test suite T exercising the methods of the target class C, and *(ii)* producing a set $M_1, \ldots, M_n$ of mutants of C, representing synthetic faults in the class. As Figure 4 shows, these artifacts are used at different stages of the technique. We used Randoop [41] for test generation and Major [27] for mutant generation. Although we used these tools in our current implementation, the user may replace them with other tools or even provide her own test suite and mutated versions of the target class C.

4.2 Grammar Extractor

The *Grammar Extractor* takes as input a class C and creates a grammar G_C expressing the language of candidate assertions for C. Those assertions denote method preconditions, postconditions, and class invariants. The extractor instantiates our base grammar, referred to as B, with information that is specific to C, e.g., attribute types, legally typed navigational expressions involving the attributes, etc.

Figure 5 shows a fragment of the base grammar B, capturing the fixed parts of the specification language, i.e., the parts that are common to any input class of interest. For this paper, the grammar B supports numerical comparisons, logical expressions, membership expressions, and quantified expressions. Numerical comparisons and

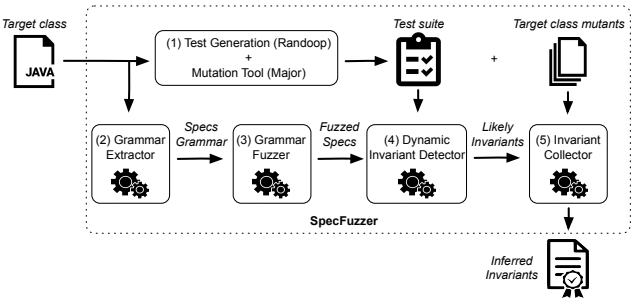

Figure 4: The SpecFuzzer workflow.

[2] Although our approach is general and language independent, some parts of our current prototype, such as the grammar extraction and the evaluation of candidate (fuzzed) specifications, are currently implemented for Java. Supporting other languages that Daikon can handle, like C, would require the implementation of such parts.

⟨*FuzzedSpec*⟩ ::= ⟨*QuantifiedExpr*⟩ | ⟨*BooleanExpr*⟩

⟨*QuantifiedExpr*⟩ ::= ⟨*Quantifier*⟩ ⟨*Typed_Var*⟩ '`:`' ⟨*BooleanExpr*⟩

⟨*Quantifier*⟩ ::= '`all`' | '`exists`'

⟨*BooleanExpr*⟩ ::= ⟨*NumCmpExpr*⟩ | ⟨*LogicCmpExpr*⟩ | ⟨*MembershipExpr*⟩ | '`!`' ⟨*BooleanExpr*⟩

⟨*NumCmpExpr*⟩ ::= ⟨*NumExpr*⟩ ⟨*NumCmpOp*⟩ ⟨*NumExpr*⟩ | ⟨*NumExpr*⟩ ⟨*NumCmpOp*⟩ ⟨*NumExpr*⟩ ⟨*NumBinOp*⟩ ⟨*NumExpr*⟩

⟨*NumExpr*⟩ ::= ⟨*NumVar*⟩ | ⟨*NumConst*⟩

⟨*LogicCmpExpr*⟩ ::= ⟨*BooleanExpr*⟩ ⟨*LogicOp*⟩ ⟨*NumCmpExpr*⟩ | '`(`' ⟨*BoolVar*⟩ ⟨*LogicOp*⟩ ⟨*BoolVar*⟩ '`)`' ⟨*LogicOp*⟩ ⟨*NumCmpExpr*⟩ | '`(`' ⟨*NumCmpExpr*⟩ '`)`' ⟨*LogicOp*⟩ '`(`' ⟨*NumCmpExpr*⟩ '`)`'

⟨*MembershipExpr*⟩ ::= ⟨*type_SetExpr*⟩.has(⟨*type_Var*⟩)

⟨*NumCmpOp*⟩ ::= '`==`' | '`!=`' | '`>`' | '`<`' | '`<=`' | '`>=`'

⟨*NumBinOp*⟩ ::= '`+`' | '`-`' | '`*`' | '`/`' | '`%`'

⟨*LogicOp*⟩ ::= '`||`' | '`xor`' | '`==>`' | '`<==>`'

Figure 5: Fragment of the base grammar B.

logical expressions are the simplest constructs of the language. They relate numerical expressions and boolean expressions by using traditional numerical operators and logical connectives—⟨*NumCmpOp*⟩ and ⟨*LogicOp*⟩, respectively. Membership expressions allow one to express whether or not a typed element belongs to a set (collection) of the corresponding type. The grammar fragment uses the `has` notation from JML, and shows a production rule for typed variables. Although it is not explicitly shown in the grammar fragment, the `reach` operator is a way of building a typed set expression. A concrete example of a membership expression from a formula shown in Section 3 is the following:

```
reach(this, next).has(l)
```

expressing that list `l` belongs to the set of objects reachable from `this` by navigations (zero or more) through `next`. Finally, the grammar allows for existential and universal quantification. Again, an example of a quantified expression from Section 3 is the following:

```
exists SortedList l:
    reach(this, next).has(l) && l.elem == SENTINEL
```

whose intuitive reading is that there exists a list object reachable from `this` with the field `elem` holding the `SENTINEL`.

To obtain the grammar G_C, the grammar extractor takes B and adds or deletes symbols and production rules, based on the structure of C. The process basically depends on C's direct and indirect fields (fields declared in C itself or in a class reachable from C). Intuitively, from every field/navigation, a terminal symbol of the corresponding type is defined (e.g., `this.next` will be a terminal of type `SortedList`).

Set expressions deserve a more detailed description. Firstly, if a field `f` is of a `Collection` type, then `f` will be a terminal of (typed) `SetExpr`. For instance, if `SortedList` were an implementation of `Collection`, then `this` and `this.next` would be terminals of type `SortedList_SetExpr`. Secondly, the `reach` operator is also involved in building set expressions. For expression `e` and recursive field `f` (a field is recursive if it is defined in a class `C` and has type `C`) of class `C`, a production rule allows expression `reach(e, f)` to have type `C_SetExpr`. Thus, expression `reach(this, next)` has type `SortedList_SetExpr`.

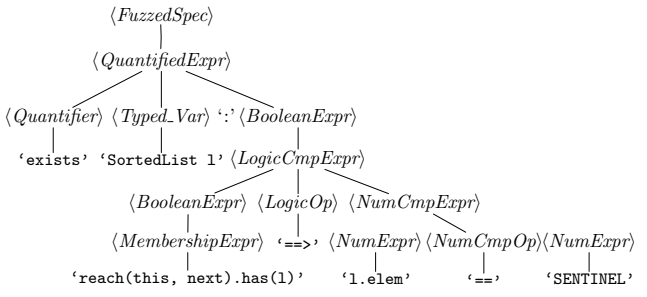

Figure 6: A derivation tree produced by our Grammar Fuzzer for the expression `exists SortedList l: reach(this, next).has(l) && l.elem == SENTINEL`.

4.3 Grammar Fuzzer

The goal of the *Grammar Fuzzer* component is to produce candidate assertions. It uses a standard generative grammar-based fuzzer to achieve this goal [25, 50]. This component produces derivations of G_c—i.e., strings in $\mathcal{L}(G_c)$—to obtain assertions for C. It begins with the start symbol ⟨*FuzzedSpec*⟩ and keeps expanding non-terminal symbols until no more non-terminals are present. Each non-terminal symbol is expanded based on a non-deterministic choice and, to avoid expansions leading to infinite derivation paths, a limit of 5 on the number of non-terminals is defined. Furthermore, to avoid getting stuck in a situation where the number of symbols cannot be reduced any further, the total number of expansion steps is also limited to 100. The rationale for this choice is that complex class assertions can be created by combining small assertions, rather than longer ones. Figure 6 shows the derivation tree of the property used in our illustrative example: `exists SortedList l: reach(this, next).has(l) && l.elem == SENTINEL`.

By using this derivation mechanism, our *Grammar Fuzzer* produces candidate predicates very efficiently. In all of our experiments we generated up to 2,000 different candidates every time we executed SpecFuzzer. Furthermore, as the grammar G_C has been specifically extracted for a class C, all the specifications generated by the fuzzer are guaranteed to express properties over C. We implemented our fuzzer in Java, reproducing a general grammar-based fuzzer written in Python [50].

4.4 Dynamic Invariant Detector

The goal of the *Dynamic Invariant Detector* is to evaluate the plausibility of the candidate assertions produced by the fuzzer. As Figure 4 shows, the dynamic invariant detector takes as input a test suite, produced by the test generator, and a set of assertions, produced by the fuzzer. This component instruments the program with the assertions generated by the fuzzer and runs the tests to verify which assertions hold across all executions. The resulting assertions are reported as *likely invariants*.

The dynamic invariant detector is built on top of Daikon,[3] a state-of-the-art tool for likely invariant detection [18]. We used Daikon as follows. We configured Daikon to include the assertions we provided—i.e., the expressions produced by the fuzzer—in the initial pool of candidate assertions it uses. For that, we used a mechanism provided by Daikon to incorporate new constraints.[4] Furthermore, we included, along with the new constraints, a component that allows the tool to interpret and evaluate the assertions at run-time.

4.5 Invariant Selector

The goal of the *Invariant Selector* is to partition the assertions that were deemed valid by the dynamic invariant detector, grouping together similar assertions, and taking a single representative from each partition. At the same time, this component discards assertions that, although were confirmed by the invariant detector, are considered weak and thus less relevant. This component takes as input the set of likely invariants obtained from the previous step, and the set of mutants of the input class `C` (obtained from a mutation tool). This component reports a subset of the likely assertions it receives as input, ranking the invariants by the number of failures in corresponding code assertions. The Invariant Selector reduces the number of reported assertions. To sum up, this component discards an assertion because of one of two reasons:

(1) the assertion is considered *weak*, and not to capture relevant properties of the target class;
(2) the assertion is (semantically) *similar* to another produced assertion.

In the following, we describe how we approximate the detection of weak and equivalent specifications via mutation analysis.

4.5.1 Detecting weak specifications with Mutation Analysis. Recall that the fuzzer reports thousands of constraints and the dynamic invariant detector (in our case, Daikon) can only discard specifications invalidated by the tests. Several constraints can still "survive" the filtering process described on Section 4.4. Even with better test suites some assertions would still "survive" that process. For example a *tautology*, such as `x >= y || x <= y`, would *not* be invalidated by Daikon as it is a valid proposition, but it is unlikely to be useful. Being syntactically driven, the fuzzer can produce valid assertions that do not provide any interesting information. The assumption is that it also generates interesting ones. SpecFuzzer uses *mutation analysis to discard uninteresting assertions*. The idea is the following: if a likely invariant –an assertion that is not falsified by the test suite of `C`– cannot be falsified by any mutant of `C`, then it is a property that not only holds on `C`, but also on all synthetic buggy versions of `C`. We will then consider it a *weak* assertion, and discard it as being *irrelevant*. This approach of using mutation analysis to induce more effective (stronger) oracles has been used in prior work, notably by Fraser and Zeller [20], as well as in recent work on oracle improvement [47] and specification inference [38].

4.5.2 Clustering similar specifications with Mutation Analysis. It is possible that SpecFuzzer produces syntactically different assertions that are semantically equivalent (or similar with respect to a distance metric). SpecFuzzer tries to identify and *remove* such assertions. As an example, consider the following assertions that are produced by SpecFuzzer on our `SortedList` example:

```
all SortedList l: reach(this, next).has(l) ==>
                                    l.elem <= l.next.elem
all SortedList l: !(reach(this, next).has(l) &&
                                    l.elem > l.next.elem)
```

Both these assertions express the sortedness property on lists. The equivalence of these assertions follows from De Morgan's laws [40], algebraic properties of integers, and the equivalence of boolean connectives. To identify equivalent assertions and assertions that are similar with respect to their ability to capture synthetic faults, SpecFuzzer again uses mutation analysis. Two assertions will be considered similar if they kill the same set of mutants, i.e., if the are falsified on the same set of program faults. For example, the two assertions above kill the same set of 2 mutants, together with 74 other assertions. SpecFuzzer uses this mutation-based notion of assertion equivalence to partition the set of likely assertions according to the mutants they kill. Moreover, from each partition, SpecFuzzer selects a representative assertion. To do so, it proceeds with the following heuristic: the assertions in each partition are ranked by the number of times they fail when running the test suite on the mutants (while they all kill the same mutants, some assertions may fail a greater number of times, i.e., for more tests in the test suite). The rationale is that assertions that fail the most represent stronger properties, and thus they may subsume other assertions in the partition. Considering the `SortedList.insert` example, this mechanism enabled SpecFuzzer to reduce the number of reported specifications from 437 specifications to 16. Figure 7 shows the breakdown of assertions classified as irrelevant (Section 4.5.1), equivalent (Section 4.5.2), and reported, for the example.

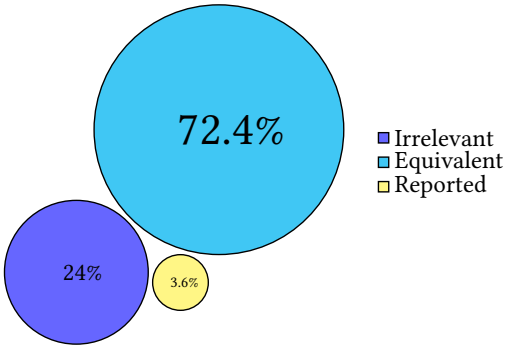

Figure 7: Breakdown of reasons for discarding specifications on `SortedList.insert`. Only 3.6% of the specifications that "survive" the invariant detection stage are reported.

Artifact. SpecFuzzer is publicly available for download [5].

5 EVALUATION

To evaluate SpecFuzzer, we performed a series of experiments focused on the following research questions:

RQ1 *Is grammar-based fuzzing effective at generating relevant assertions?*

[3] http://plse.cs.washington.edu/daikon/
[4] http://plse.cs.washington.edu/daikon/download/doc/developer.html#New-invariants

RQ2 *Is the mutation-based selector successful for removing redundant/irrelevant assertions?*

RQ3 *How does SpecFuzzer compare with alternative techniques?*

RQ1 analyzes the effectiveness of using grammar-based fuzzing as a technique to generate candidate assertions, with respect to a ground truth. RQ2 evaluates the suitability of the mutation-based assertion selection component of SpecFuzzer, at discarding weak assertions, and prioritizing the most relevant. Finally, RQ3 compares the effectiveness of SpecFuzzer with the state-of-the-art techniques GAssert [47] and EvoSpex [38].

5.1 Evaluation Subjects

The performance evaluation in previous work used indirect metrics to compute precision and recall [38, 47]. In this paper, we compute these measures directly, which requires having a ground truth for comparison, for each evaluation subject. We took all the 34 Java methods that were part of the evaluation of GAssert [47], and all 23 methods in the contract reproducibility evaluation of EvoSpex [38], obtaining a data set composed of 57 subject methods. We studied each method, and manually produced corresponding "ground truth" assertions capturing the intended behavior of the corresponding method. We made our best effort to be fair in the construction of this ground truth, both in capturing as much of the ground truth as possible, and in how the ground truth is modularized as a set of properties (each ground truth specification is expressed as a conjunction of assertions). They were all cross-checked by authors, and when possible, their validity was verified using SMT (via Microsoft IntelliTest). With this process, we obtained a total of 80 ground truth assertions, for the 57 methods. Each obtained ground truth specification has between 1 and 3 conjuncts. The details about these assertions can be found in our replication package.

Notice that, since previous work focuses only on inferring postconditions, our evaluation also focuses on these program points, although SpecFuzzer can infer assertions for various program points. After obtaining the ground truth composed of 80 assertions, we carefully examined each ground truth assertion, to determine if it can be expressed in the assertion language of at least one of the evaluated tools. Since 15 of the 80 assertions were not supported by any of these tools, we discarded them[5]. After that, we also removed methods that ended up without ground truth postconditions. This results in 65 postcondition assertions, for 43 Java methods.

Each assertion within the 65 in the ground truth could be expressed in the assertion language of at least one tool. GAssert's language can express 28 of the 65; EvoSpex's language can express 29 of the 65; and SpecFuzzer can express 41 of the 65. Although our grammar supports all 65 assertions, when implementing support for the grammar into Daikon's assertion template instantiation, some expressions are ignored by Daikon's infrastructure (e.g., expressions that require instantiating templates with objects of different classes). There is no fundamental reason why these issues cannot be resolved, but they demand substantial modifications in Daikon.

[5]Discarded assertions include complex trigonometric properties, vector cross product constraints, assertions involving characters and strings, and conversions between characters and hexadecimal encodings.

Table 3: Inferring assertions with grammar-based fuzzing.

Ground-truth	#Reported	#Detected	%Detected
65	20,277	40	**61.5**
41	15,555	40	**97.5**

5.2 Experimental Setup

SpecFuzzer requires a test suite for the class under analysis, and a set of mutants for this class. The test suite was generated using Randoop [41], and the tool was instructed to generate a maximum of 500 test sequences. Mutants were generated using Major [27], with all supported mutation operators enabled. The fuzzer was run until 2,000 different candidate assertions were generated (syntactic duplicates were removed), for each subject class.

Regarding GAssert and EvoSpex, we followed the same methodology described in the corresponding papers [38, 47], using exactly the same configuration parameters for the evolutionary processes of each technique. Moreover, to account for the randomness of each approach, for each of the 43 Java methods, we ran each of the tools to infer postconditions a total of 10 times. All the results reported in this section correspond to the averages of the executions.

We set a timeout of 90 minutes, for each execution of each tool. All tools were run on an Intel Core i7 3.2Ghz, with 16Gb of RAM, running GNU/Linux (Ubuntu 18.04). The detailed description of how to reproduce the experiments in this paper is available in the replication package site.

5.3 Effectiveness of Grammar-based Fuzzing

The effectiveness of grammar-based fuzzing in producing relevant assertions is measured against assertions in the ground truth. The experiment for RQ1 consisted in running SpecFuzzer on each subject, and analyzing the percentage of those assertions that the tool was able to infer. Recall that the invariant selector uses a (mutation-based) heuristic to discard assertions. As such, it may wrongly discard relevant assertions. For that reason, to answer RQ1, we ran SpecFuzzer with the invariant selector disabled.

We manually inspected the output of SpecFuzzer. More precisely, we manually analyzed the assertions that SpecFuzzer reports to verify if they were present in the ground truth (and if the ground truth assertions were present in the output as well). In some cases, it was difficult to determine if a given assertion was equivalent to a certain assertion in the ground truth. When the obtained expressions allowed for it, we used an SMT solver via Microsoft IntelliTest to check expression equivalence. More precisely, we produced C# programs whose branch conditions captured implication and equivalence between output candidate assertions and ground truth assertions, and used the dynamic symbolic execution tool IntelliTest to check whether such expressions could be falsified.

Table 3 summarizes the results of the experiments for RQ1, with respect to the overall ground truth (65 assertions) and the subset of the ground truth that is actually supported by SpecFuzzer (as we explained before, 41 out of the 65 are currently supported by our implementation). We report the number of reported assertions, the number of ground truth assertions detected by the tool, and the percentage of ground truth assertions that were detected. If we consider the language supported by our implementation, our tool

Table 4: Performance of the Invariant Selector reducing assertions.

Subject	#G	#Reported		Detected(%)		Red. (%)
		Before	After	Before	After	
oasis.SimpleMethods	4	115	31	75	75	73
daikon.StackAr	8	2067	70	87.5	62.5	96.6
daikon.QueueAr	8	4699	152	50	50	96.7
math.ArithmeticUtils	1	4	2	100	100	50
math.FastMath	2	60	31	100	100	48.3
math.MathUtils	1	19	4	0	0	78.9
lang.BooleanUtils	5	49	12	100	100	75.5
guava.IntMath	1	314	46	0	0	85.3
tsuite.Angle	2	3	0	50	0	100
tsuite.MathUtil	3	22	13	33.3	33. 3	40.9
tsuite.Envelope	1	1094	27	0	0	97.5
eiffel.Composite	4	8696	42	75	0	99.5
eiffel.DLLN	3	137	29	100	100	78.8
eiffel.Map	6	140	22	16.6	16.6	84.2
eiffel.RingBuffer	5	1947	269	20	20	86.1
cozy.Polyupdate	3	382	119	66.6	66.6	68.8
cozy.Structure	2	153	21	100	100	86.2
cozy.ListComp02	2	62	5	0	0	91.9
cozy.MinFinder	1	17	2	100	100	88.2
cozy.MaxBag	3	297	78	100	100	73.7
TOTAL	65	20277	975	61.5	52.3	95.1
	41	15555	618	97.5	82.9	96

correctly detects 97.5% of the assertions in the ground truth; if we consider the language supported by at least one of the specification inference tools, SpecFuzzer correctly detects 61.5% of the ground truth assertions. These results confirm that grammar-based fuzzing is effective in generating relevant assertions (as shown later on, even when considering the 65 assertions in the ground truth, the performance of fuzzing is comparable with state-of-the-art tools).

5.4 Performance of Invariant Selection

The invariant selector component of SpecFuzzer implements a mutation-based heuristics to reduce the number of reported assertions, discarding "weak" assertions (assertions that survive all mutants), and selecting representatives among "similar" assertions (assertions that kill the same mutants). RQ2 evaluates the performance of this stage. The experiment in this case compares the assertions obtained after invariant detection, with the assertions that are preserved after running the invariant selection. The comparison measures assertion reduction, and the percentage of the ground truth that is covered prior and after assertion selection.

Table 4 shows these results for all subjects, grouped by class name. In each case we report assertions in the ground truth, reported assertions before and after invariant selection, and percentage of the ground truth that is covered, again, before and after invariant selection. Finally, we indicate the reduction rate obtained by invariant selection (number of assertions after selection, with respect to the number of assertions before selection).

The invariant selection results show that the mutation-based heuristics in SpecFuzzer effectively reduces the number of reported assertions, with a relatively small loss in property detection (with respect to the ground truth). More precisely, the reported assertions are reduced by ~95%, and 6 out of the 40 correctly fuzzed assertions are discarded (covering ~52% of the ground truth of 65 assertions, ~83% of the 41 ground truth assertions that the tool supports).

Table 5: Valid assertions discarded by the Invariant Selector.

Subject	Assertion
StackAr.pop	`theArray[old(top)] == null`
StackAr.topAndPop	`theArray[old(top)] == null`
Angle.getTurn	`abs(res) <= 1`
Composite.addChild	`c.value == old(c.value))`
	`children == old(children)`
	`ancestors == old(ancestors)`

To understand the reasons why we miss 6 ground truth assertions during invariant selection, we analyzed how these assertions are classified by the detector. In all cases, the assertions are deemed as *irrelevant*, i.e., they are not killed by any mutant. While the problem may be a weak test suite, it becomes clear, when observing the assertions, that there is no mutation operator able to kill these assertions (the assertions are shown in Table 5). The problem is not specific to Major (the mutation tool that we used); other tools such as PIT do not have mutants able to kill these assertions either. Let us provide two concrete examples. Assertion `abs(res) <= 1` for `Angle.getTurn` corresponds to a method whose `result` is either 0, -1 or 1; no mutant makes this method return a value other than these. In the `Composite.addChild` subject, assertion `c.value == old(c.value)` would be violated if a mutant changed the value of `c`, a parameter of the method; a mutation operator achieving this effect would have to add a new sentence.

These observations suggest that we may improve the effectiveness of our heuristics by extending Major with support for additional mutation operators, specific to our purposes.

5.5 Comparison of GAssert, EvoSpex and SpecFuzzer

RQ3 compares SpecFuzzer with the state-of-the-art tools GAssert and EvoSpex. The comparison is based on standard performance metrics: precision, recall and f1-score. These metrics are computed with respect to the ground truth that we produced for the evaluation subjects, as follows. Given a set G of ground truth formulas, the precision of a set A of assertions produced by a tool is computed by determining the number of assertions in A that are implied by G. Many assertions were trivially incorrect (not implied by the ground truth), and were manually identified. In more complex cases, (in)correctness was determined using IntelliTest. Once the set I of incorrect assertions in A was determined, precision is computed with the formula $(\#A - \#I)/\#A$. To compute recall, we check the number of formulas in G that are implied by $(A - I)$. Again, while some cases were trivial to check manually, for instance, when a ground truth formula was directly present in $A - I$, more complex ones were confirmed using IntelliTest. Recall is then computed by the formula $\#N/\#G$, where N is the set of ground truth properties implied by $A - I$.

Tools were run to infer assertions as described earlier in this section, and the results are shown in Table 6, grouped by subject class. Columns #M and #G show the number of methods in the subject and the number of assertions in the ground truth, respectively. For each technique we show the number of inferred assertions, the precision and recall with respect to the ground truth, and the f1-score. We

Table 6: Precision, Recall and F1-Score of GAssert, EvoSpex and SpecFuzzer on the data set.

Subject	#M	#G	#Inferred			Precision(%)			Recall(%)			F1-Score		
			GAssert	EvoSpex	SpecFuzzer	GAssert	EvoSpex	SpecFuzzer	GAssert	EvoSpex	SpecFuzzer	GAssert	EvoSpex	SpecFuzzer
oasis.SimpleMethods	4	4	7	4	31	100	75	100	50	25	75	0.66	0.37	0.85
daikon.StackAr	5	8	5	6	70	100	83.3	87.1	37.5	37.5	62.5	0.54	0.51	0.72
daikon.QueueAr	5	8	8	12	152	100	91.6	61.1	37.5	25	50	0.54	0.39	0.54
math.ArithmeticUtils	1	1	1	0	2	100	100	50	100	0	100	1	0	0.66
math.FastMath	1	2	3	1	31	100	0	61.2	100	0	100	1	0	0.75
math.MathUtils	1	1	1	1	4	100	0	100	0	0	0	0	0	0
lang.BooleanUtils	2	5	2	2	12	50	100	83.3	0	0	100	0	0	0.9
guava.IntMath	1	1	3	3	46	100	66.6	97.8	100	0	0	1	0	0
tsuite.Angle	1	2	3	1	0	100	100	100	0	0	0	0	0	0
tsuite.MathUtil	1	3	3	1	13	100	100	84.6	33.3	0	33.3	0.49	0	0.47
tsuite.Envelope	1	1	3	4	27	100	100	18.5	0	0	0	0	0	0
eiffel.Composite	1	4	0	7	42	100	100	50	0	50	0	0	0.66	0
eiffel.DLLN	2	3	0	4	29	100	100	89.6	0	66.6	100	0	0.79	0.94
eiffel.Map	3	6	4	10	22	50	100	81.8	16.6	66.6	16.6	0.24	0.79	0.27
eiffel.RingBuffer	5	5	9	31	269	88.8	87	64.6	20	40	20	0.32	0.54	0.3
cozy.Polyupdate	2	3	3	3	119	66.6	100	1.6	33.3	66.6	66.6	0.44	0.79	0.03
cozy.Structure	2	2	2	2	21	100	100	95.2	100	100	100	1	1	0.97
cozy.ListComp02	2	2	0	4	5	100	100	83.3	0	100	0	0	1	0
cozy.MinFinder	1	1	0	2	2	100	100	100	0	100	100	0	1	1
cozy.MaxBag	3	3	8	33	78	100	84.8	94.8	0	66.6	100	0	0.74	0.97
Total-65	43	65	65	131	975	92.3	88.5	65.8	27.6	38.4	52.3	0.42	0.53	0.57
Total-SpecFuzzer	30	41	43	76	618	95.3	88	62.4	41.4	43.9	82.9	0.57	0.58	0.71
Total-GAssert	23	28	39	54	615	94.8	85.1	63.4	64.2	46.4	71.4	0.76	0.60	0.67
Total-EvoSpex	24	29	34	82	624	91.1	92.8	59.9	44.8	86.2	72.4	0.60	0.89	0.65

summarize the performance metrics for the overall ground truth of 65 assertions, as well as in the context of assertions that are supported by each particular tool (recall that GAssert supports in its language 28 of the 65, EvoSpex 29 of the 65, and SpecFuzzer 41 of the 65). That is, rows Total-SpecFuzzer, Total-GAssert and Total-EvoSpex show the performance of the techniques on the portion of the ground truth that SpecFuzzer, GAssert and EvoSpex support, respectively.

Inferred Assertions. If we focus on the number of inferred assertions, GAssert and EvoSpex report fewer assertions than SpecFuzzer. This is an advantage of the previous techniques, since the produced output is easier to interpret. The main reason here is that both techniques feature evolutionary processes, that aim at minimizing the size of the assertions (this is an objective of both evolution processes). SpecFuzzer is in this respect a simpler technique. Still, the invariant selector component allows our tool to report a reasonable number of assertions (22 per method, on average). This number is still large, and calls for future work to further reduce the number of assertions that SpecFuzzer reports. A possible approach is to exploit the mutation killing information to identify subsumption/implication relations across assertions, so that only the stronger assertions are reported. More precisely, a mutation-based notion of implication would consider that an assertion α_1 implies another α_2 if the set of mutants killed by α_1 includes those killed by α_2. It is important to notice that as our assertion equivalence definition and mutation-based identification of weak assertions are approximate, so our current assertion reduction mechanism already affects recall (Table 4 shows some concrete examples). More aggressive reduction mechanisms, such as the above described based on subsumption/implication, may affect recall even further. A way of reducing reported assertions without compromising recall would require precise equivalence/implication checking across assertions (e.g., using SAT or SMT). Although this is a viable option, it may considerably affect both efficiency and scalability, and thus the generality of the technique.

Precision. Precision is the aspect in which GAssert and EvoSpex outperform SpecFuzzer. Again, this has to do with the fact that both GAssert and EvoSpex incorporate mechanisms to actively reduce the number of false positives (understood as invalid properties). In particular, GAssert iteratively improves assertions using OASIs [26], launching EvoSuite instances to search and detect defects in the candidate assertions. EvoSpex uses a bounded-exhaustive test generation technique with the aim of building a more thorough test suite, able to discard more false positives. Both techniques have disadvantages associated with these processes. GAssert pays a price in efficiency (it is the most costly of the three); EvoSpex's bounded exhaustive test generation has scalability issues (due to its bounded exhaustive test generation, it has difficulties scaling to larger subjects).

SpecFuzzer borrows from Daikon the mechanism to deal with precision. This issue can be dealt with by improving test suite quality. We used Randoop in our experiments, which may be complemented by additional automated test generation techniques.

Recall. Recall is the aspect where SpecFuzzer outperforms GAssert and EvoSpex. This is the case for the overall ground truth, and for

most tool-specific ground truth subsets. EvoSpex has better recall than SpecFuzzer for its specific language; it infers 5 assertions that SpecFuzzer cannot. Out of these 5, 2 assertions are discarded by the invariant selector (we described the reasons above). The remaining three are supported by the grammar, but Daikon is currently unable to instantiate these assertions (we also described this issue earlier in the paper); that is, these three assertions are not part of the 41 that our prototype currently supports.

The recall improvement of SpecFuzzer over the other techniques makes our tool more effective overall, as summarized by the f1-scores. Notice that SpecFuzzer has a better f1-score compared with previous techniques, for the overall ground truth, i.e., even taking into account its precision limitations, and the current issues with support for assertions.

5.6 Threats to Validity

Our experimental evaluation was performed on a data set built with subjects from previous works. We needed to manually study these subjects in order to define the ground truth assertions. To mitigate the risk of errors, we checked these assertions using Microsoft IntelliTest (previously named Pex [48]).

Threats to internal validity may arise from the randomness of the each technique. To account for this issue, we evaluated SpecFuzzer, GAssert and EvoSpex over multiple runs on each subject method, and reported the averages. As further work, we plan to extend the experimental evaluation to larger-scale Java projects, which will likely imply abandoning the computation of performance metrics over ground truths, due to the effort that would involve studying larger projects and manually writing correct assertions.

6 RELATED WORK

The use of assertions in programs has a long tradition. Originally, assertions were used as part of approaches for software verification [24], and soon were incorporated into programming languages, for run-time checking [12]. Assertions are currently used for multiple software development activities: program verification [11, 17, 19, 21, 22, 30], software design [36], bug finding [30, 31, 41, 48], program comprehension and maintenance [45], program repair [16, 34, 43, 44], among others.

Specification inference is an active area of research. Besides the techniques that infer contract assertions, with which we have compared our technique in this paper [38, 47], other related approaches exist, in particular for inferring test oracles [20, 49] (that is, assertions that are valid only for specific unit tests), and other kinds of specifications, such as behavioral descriptions [14, 28, 29]. These techniques seek related but different objectives, and thus can complement each other. In relation to test assertion inference, tools and techniques for inferring test assertions produce specifications that are difficult to generalize as contracts; contract specifications, on the other hand, can be instantiated as test assertions, but may capture weaker properties, compared to their test assertion counterparts. Other related techniques attempt to produce assertions from other sources, such as comments [9], or weaker forms of specifications, notably metamorphic relations [10]. Other related forms of specification inference focus on different properties, e.g., behavioral properties in linear-time temporal logic [32], or properties that describe the temporal relationships between different methods in an API [15]. As described in [32], these techniques that infer behavioral properties can be complemented by Daikon [18] (the dynamic invariant detection technique that our work is based on), and therefore they can also profit from more expressive assertions.

Previous approaches have worked on improving Daikon's effectiveness. In particular, the work reported in [39] combines Daikon with static verification, in a way that can be understood as an improvement to precision (static verification is employed to confirm assertion validity). Our approach, on the other hand, is largely motivated by automatically equipping Daikon with more expressive assertions, an issue not tackled in [39]. We are not aware of other approaches that automatically address the expressiveness limitations of Daikon. Fuzzing [50] is also a very active topic, with known applications in security vulnerability discovery, and bug finding in general. To the best of our knowledge, our approach is the first to employ fuzzing to produce candidate formal specifications.

7 CONCLUSION AND FUTURE WORK

Formal class specifications have applications in various areas of software development, including software design, bug finding, and program comprehension. Techniques to automatically infer class specifications have been proposed, but are limited, e.g., they support a limited number of assertion types and are inflexible to change. To fill this gap, we presented SpecFuzzer, a technique to infer likely class specifications that combines static analysis, grammar-based fuzzing, and mutation analysis. Our evaluation shows that SpecFuzzer has superior performance in comparison with the state-of-the-art tools GAssert and EvoSpex, especially considering recall. Furthermore, the use of grammar-based fuzzing enables SpecFuzzer to be easily adapted to different assertion languages.

This paper also opens various lines for improvement, as we have identified some concrete limitations of our approach. The mutation-based mechanisms to cluster equivalent assertions and discard weak assertions are affected by the absence of mutation operators, that would allow our tool to detect some specific constraints. Other more sophisticated mechanisms to deal with assertions not killed by any mutant may also be incorporated (e.g., constraint-based techniques). In general, the modular structure of our technique enables us to improve specific components, e.g., test generation (to improve precision), fuzzing (to consider more effective/efficient fuzzing techniques), etc. Finally, implementation limitations in the dynamic detection phase constitute a bottleneck for SpecFuzzer's assertion inference capabilities, that we plan to address in future extensions of our tool.

ACKNOWLEDGEMENTS

This work is partially supported by INES (www.ines.org.br); CNPq grant 465614/2014-0; CAPES grant 88887.136410/2017-00; FACEPE grants APQ-0399-1.03/17 and PRONEX APQ/0388-1.03/14; and ANPCyT grants PICT 2017-2622 and PICT 2019-2050. Facundo Molina's work is also supported by Microsoft Research, through a Latin America PhD Award.

Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring entities.

REFERENCES

[1] 2021. Fuzzing Research Tools and their Relatedness. https://fuzzing-survey.org/

[2] 2021. Google's AFL. https://github.com/google/AFL

[3] 2021. Google's ClusterFuzz. https://github.com/google/clusterfuzz

[4] 2021. Google's OSS-Fuzz. https://github.com/google/oss-fuzz

[5] 2022. SpecFuzzer implementation and replication package. https://sites.google.com/view/specfuzzer

[6] Pablo Abad, Nazareno Aguirre, Valeria S. Bengolea, Daniel Alfredo Ciolek, Marcelo F. Frias, Juan P. Galeotti, Tom Maibaum, Mariano M. Moscato, Nicolás Rosner, and Ignacio Vissani. 2013. Improving Test Generation under Rich Contracts by Tight Bounds and Incremental SAT Solving. In *Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, March 18-22, 2013*. IEEE Computer Society, 21–30. https://doi.org/10.1109/ICST.2013.46

[7] Paul Ammann and Jeff Offutt. 2008. *Introduction to Software Testing*. Cambridge University Press. https://doi.org/10.1017/CBO9780511809163

[8] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Trans. Software Eng.* 41, 5 (2015), 507–525. https://doi.org/10.1109/TSE.2014.2372785

[9] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. 2018. Translating code comments to procedure specifications. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, Frank Tip and Eric Bodden (Eds.). ACM, 242–253. https://doi.org/10.1145/3213846.3213872

[10] Arianna Blasi, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Antonio Carzaniga. 2021. MeMo: Automatically identifying metamorphic relations in Javadoc comments for test automation. *J. Syst. Softw.* 181 (2021), 111041. https://doi.org/10.1016/j.jss.2021.111041

[11] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. 2005. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*. 342–363. https://doi.org/10.1007/11804192_16

[12] Lori A. Clarke and David S. Rosenblum. 2006. A historical perspective on runtime assertion checking in software development. *ACM SIGSOFT Softw. Eng. Notes* 31, 3 (2006), 25–37. https://doi.org/10.1145/1127878.1127900

[13] Marcelo d'Amorim, Carlos Pacheco, Tao Xie, Darko Marinov, and Michael D. Ernst. 2006. An Empirical Comparison of Automated Generation and Classification Techniques for Object-Oriented Unit Testing. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), 18-22 September 2006, Tokyo, Japan*. IEEE Computer Society, 59–68. https://doi.org/10.1109/ASE.2006.13

[14] Guido de Caso, Víctor A. Braberman, Diego Garbervetsky, and Sebastián Uchitel. 2013. Enabledness-based program abstractions for behavior validation. *ACM Trans. Softw. Eng. Methodol.* 22, 3 (2013), 25:1–25:46. https://doi.org/10.1145/2491509.2491519

[15] Guido de Caso, Víctor A. Braberman, Diego Garbervetsky, and Sebastián Uchitel. 2013. Enabledness-based program abstractions for behavior validation. *ACM Trans. Softw. Eng. Methodol.* 22, 3 (2013), 25:1–25:46. https://doi.org/10.1145/2491509.2491519

[16] Brian Demsky, Michael D. Ernst, Philip J. Guo, Stephen McCamant, Jeff H. Perkins, and Martin Rinard. 2006. Inference and Enforcement of Data Structure Consistency Specifications. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis* (Portland, Maine, USA) *(ISSTA '06)*. Association for Computing Machinery, New York, NY, USA, 233–244. https://doi.org/10.1145/1146238.1146266

[17] Greg Dennis, Felix Sheng-Ho Chang, and Daniel Jackson. 2006. Modular verification of code with SAT. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006*, Lori L. Pollock and Mauro Pezzè (Eds.). ACM, 109–120. https://doi.org/10.1145/1146238.1146251

[18] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.* 69, 1-3 (2007), 35–45. https://doi.org/10.1016/j.scico.2007.01.015

[19] Manuel Fähndrich. 2010. Static Verification for Code Contracts. In *Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings*. 2–5. https://doi.org/10.1007/978-3-642-15769-1_2

[20] Gordon Fraser and Andreas Zeller. 2010. Mutation-driven generation of unit tests and oracles. In *ISSTA*. ACM, 147–158.

[21] Carlo A. Furia, Martin Nordio, Nadia Polikarpova, and Julian Tschannen. 2017. AutoProof: auto-active functional verification of object-oriented programs. *Int. J. Softw. Tools Technol. Transf.* 19, 6 (2017), 697–716. https://doi.org/10.1007/s10009-016-0419-0

[22] Juan P. Galeotti, Nicolás Rosner, Carlos López Pombo, and Marcelo F. Frias. 2010. Analysis of invariants for efficient bounded verification. In *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010*, Paolo Tonella and Alessandro Orso (Eds.). ACM, 25–36. https://doi.org/10.1145/1831708.1831712

[23] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. 2002. *Fundamentals of Software Engineering* (2nd ed.). Prentice Hall PTR, Upper Saddle River, NJ, USA.

[24] C. A. R. Hoare. 2003. Assertions: A Personal Perspective. *IEEE Ann. Hist. Comput.* 25, 2 (2003), 14–25. https://doi.org/10.1109/MAHC.2003.1203056

[25] 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* (Lake Buena Vista, FL, USA) *(A-TEST 2018)*. Association for Computing Machinery, New York, NY, USA, 45–48. https://doi.org/10.1145/3278186.3278193

[26] Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella. 2016. Test oracle assessment and improvement. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, Andreas Zeller and Abhik Roychoudhury (Eds.). ACM, 247–258. https://doi.org/10.1145/2931037.2931062

[27] René Just, Franz Schweiggert, and Gregory M. Kapfhammer. 2011. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, Perry Alexander, Corina S. Pasareanu, and John G. Hosking (Eds.). IEEE Computer Society, 612–615. https://doi.org/10.1109/ASE.2011.6100138

[28] Hong Jin Kang and David Lo. 2021. Adversarial Specification Mining. *ACM Trans. Softw. Eng. Methodol.* 30, 2 (2021), 16:1–16:40. https://doi.org/10.1145/3424307

[29] Tien-Duy B. Le and David Lo. 2018. Deep specification mining. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, Frank Tip and Eric Bodden (Eds.). ACM, 106–117. https://doi.org/10.1145/3213846.3213876

[30] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. 2005. How the design of JML accommodates both runtime assertion checking and formal verification. *Sci. Comput. Program.* 55, 1-3 (2005), 185–208. https://doi.org/10.1016/j.scico.2004.05.015

[31] Andreas Leitner, Ilinca Ciupa, Manuel Oriol, Bertrand Meyer, and Arno Fiva. 2007. Contract driven development = test driven development - writing test cases. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, Ivica Crnkovic and Antonia Bertolino (Eds.). ACM, 425–434. https://doi.org/10.1145/1287624.1287685

[32] Caroline Lemieux, Dennis Park, and Ivan Beschastnikh. 2015. General LTL Specification Mining (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, Myra B. Cohen, Lars Grunske, and Michael Whalen (Eds.). IEEE Computer Society, 81–92. https://doi.org/10.1109/ASE.2015.71

[33] Lisa (Ling) Liu, Bertrand Meyer, and Bernd Schoeller. 2007. Using Contracts and Boolean Queries to Improve the Quality of Automatic Test Generation. In *Tests and Proofs - 1st International Conference, TAP 2007, Zurich, Switzerland, February 12-13, 2007. Revised Papers (Lecture Notes in Computer Science, Vol. 4454)*, Yuri Gurevich and Bertrand Meyer (Eds.). Springer, 114–130. https://doi.org/10.1007/978-3-540-73770-4_7

[34] Francesco Logozzo and Thomas Ball. 2012. Modular and verified automatic program repair. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, Gary T. Leavens and Matthew B. Dwyer (Eds.). ACM, 133–146. https://doi.org/10.1145/2384616.2384626

[35] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2021. The Art, Science, and Engineering of Fuzzing: A Survey. *IEEE Trans. Software Eng.* 47, 11 (2021), 2312–2331. https://doi.org/10.1109/TSE.2019.2946563

[36] Bertrand Meyer. 1997. *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall.

[37] Barton P. Miller. 2021. Fuzz Testing of Application Reliability. http://pages.cs.wisc.edu/~bart/fuzz/

[38] Facundo Molina, Pablo Ponzio, Nazareno Aguirre, and Marcelo F. Frias. 2021. EvoSpex: An Evolutionary Algorithm for Learning Postconditions. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 1223–1235. https://doi.org/10.1109/ICSE43902.2021.00112

[39] Jeremy W. Nimmer and Michael D. Ernst. 2002. Automatic generation of program specifications. In *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2002, Roma, Italy, July 22-24, 2002*, Phyllis G. Frankl (Ed.). ACM, 229–239. https://doi.org/10.1145/566172.566213

[40] P. S. Novikov. 1964. *Elements of Mathematical Logic*. Reading, Mass., Addison-Wesley Pub. Co.

[41] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *29th International Conference on*

Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007. IEEE Computer Society, 75–84. https://doi.org/10.1109/ICSE.2007.37

[42] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Chapter Six - Mutation Testing Advances: An Analysis and Survey. *Adv. Comput.* 112 (2019), 275–378. https://doi.org/10.1016/bs.adcom.2018.03.015

[43] Yu Pei, Carlo A. Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller. 2014. Automated Fixing of Programs with Contracts. *IEEE Trans. Software Eng.* 40, 5 (2014), 427–449. https://doi.org/10.1109/TSE.2014.2312918

[44] Jeff H. Perkins, Sunghun Kim, Samuel Larsen, Saman P. Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin C. Rinard. 2009. Automatically patching errors in deployed software. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, Jeanna Neefe Matthews and Thomas E. Anderson (Eds.). ACM, 87–102. https://doi.org/10.1145/1629575.1629585

[45] Manoranjan Satpathy, Nils T. Siebel, and Daniel Rodríguez. 2004. Assertions in Object Oriented Software Maintenance: Analysis and a Case Study. In *20th International Conference on Software Maintenance (ICSM 2004), 11-17 September 2004, Chicago, IL, USA*. IEEE Computer Society, 124–135. https://doi.org/10.1109/ICSM.2004.1357797

[46] Todd W. Schiller, Kellen Donohue, Forrest Coward, and Michael D. Ernst. 2014. Case Studies and Tools for Contract Specifications. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 596–607. https://doi.org/10.1145/2568225.2568285

[47] Valerio Terragni, Gunel Jahangirova, Paolo Tonella, and Mauro Pezzè. 2020. Evolutionary Improvement of Assertion Oracles. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1178–1189. https://doi.org/10.1145/3368089.3409758

[48] Nikolai Tillmann and Jonathan de Halleux. 2008. Pex-White Box Test Generation for .NET. In *Tests and Proofs - 2nd International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4966)*, Bernhard Beckert and Reiner Hähnle (Eds.). Springer, 134–153. https://doi.org/10.1007/978-3-540-79124-9_10

[49] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. 2020. On learning meaningful assert statements for unit test cases. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 1398–1409. https://doi.org/10.1145/3377811.3380429

[50] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2019. The Fuzzing Book. In *The Fuzzing Book*. Saarland University. https://www.fuzzingbook.org/ Retrieved 2019-09-09 16:42:54+02:00.