Evolutionary Improvement of Assertion Oracles

Valerio Terragni
Università della Svizzera italiana
Lugano, Switzerland
valerio.terragni@usi.ch

Gunel Jahangirova
Università della Svizzera italiana
Lugano, Switzerland
gunel.jahangirova@usi.ch

Paolo Tonella
Università della Svizzera italiana
Lugano, Switzerland
paolo.tonella@usi.ch

Mauro Pezzè
Università della Svizzera italiana
Lugano, Switzerland
Schaffhausen Institute of Technology
Schaffhausen, Switzerland
mauro.pezze@usi.ch

ABSTRACT

Assertion oracles are executable boolean expressions placed inside the program that should pass (return true) for all correct executions and fail (return false) for all incorrect executions. Because designing perfect assertion oracles is difficult, assertions often fail to distinguish between correct and incorrect executions. In other words, they are prone to false positives and false negatives.

In this paper, we propose GAssert (Genetic ASSERTion improvement), the first technique to automatically improve assertion oracles. Given an assertion oracle and evidence of false positives and false negatives, GAssert implements a novel co-evolutionary algorithm that explores the space of possible assertions to identify one with fewer false positives and false negatives.

Our empirical evaluation on 34 Java methods from 7 different Java code bases shows that GAssert effectively improves assertion oracles. GAssert outperforms two baselines (random and invariant-based oracle improvement), and is comparable with and in some cases even outperformed human-improved assertions.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; **Genetic programming**.

KEYWORDS

program assertions, oracle improvement, evolutionary algorithm

ACM Reference Format:
Valerio Terragni, Gunel Jahangirova, Paolo Tonella, and Mauro Pezzè. 2020. Evolutionary Improvement of Assertion Oracles. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3368089.3409758

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

1 INTRODUCTION

Recently, we witnessed great advances in test input generation [13, 30]. However, the *oracle problem* [4] remains a major obstacle that limits the effectiveness of automatically generated test suites. Instead of generating test oracles for each automatically generated test case, one could rely on assertion oracles to expose software faults. Assertion oracles (also called program assertions) are executable boolean expressions that predicate on the values of variables at specific program points. A perfect assertion oracle passes (returns true) for all correct executions and fails (returns false) for all incorrect executions. Perfect oracles are difficult to design, and thus assertion oracles often fail to distinguish between correct and incorrect executions [25], that is, they are prone to both false positives and false negatives, which are jointly called *oracle deficiencies* [20]. A *false positive* is a correct program state in which the assertion fails (but should pass), and a *false negative* is an incorrect program state in which the assertion passes (but should fail).

Oracle deficiencies are a serious problem for both manually and automatically generated assertion oracles. In fact, invariant generators are known to generate invariants that are incomplete and imprecise when used as assertion oracles [6, 29, 40]. They are incomplete because most dynamic invariant generators, notably Daikon [10] and InvGen [17], cannot generate assertions that do not match pre-defined templates of Boolean expressions [6]. Existing invariant generators are also imprecise, because the generated invariants often do not generalize well with unseen test cases. In fact, Nguyen et al.'s and Staats et al.'s studies [29, 40] report high false positive rates for Daikon invariants.

Improving the quality of program assertions by removing oracle deficiencies is of paramount importance. It would improve the fault detection capability and reduce the false alarms of both automatically generated and manually written test cases.

Recently, Jahangirova et al. proposed OASIs [20, 21] to automatically identify oracle deficiencies. Given an assertion oracle, OASIs generates test cases and mutations that gives evidence of false positives and false negatives, respectively. This evidence is meant to support the developers in assessing and improving the oracles.

A recent study by OASIs's authors shows that the manual improvement of assertion oracles is difficult [22]. Given the oracle deficiencies detected by OASIs, for only 67% of the given assertions humans successfully removed all oracle deficiencies.

The difficulty of manually improving assertion oracles motivated us to study how to automatically improve assertions. Given an assertion oracle α and some evidence of false positives and false negatives provided by an oracle assessor (such as OASIs), we aim to automatically generate an improved assertion α' with fewer oracle deficiencies than α. While there are many techniques to automatically generate program assertions, for example, program invariants [2, 9, 15–17, 27, 35, 37, 46], automatically improving assertion oracles is an unexplored problem.

In this paper, we propose **GAssert**, ***Genetic ASSERTion improvement***, the first technique to automatically improve assertion oracles. Given an assertion oracle and its oracle deficiencies, GAssert explores the space of possible assertions to identify those with zero false positives and the lowest number of false negatives. GAssert favors assertions with zero false positives, as false alarms are known to trigger an expensive debugging process [29].

GAssert addresses the challenge of navigating a huge search space with an evolutionary approach that evolves populations of assertions by rewarding assertions with fewer deficiencies. GAssert formulates the oracle improvement problem as a multi-objective optimization problem (MOOP) [41] with three competing objectives: (i) minimizing the number of false positives, (ii) minimizing the number of false negatives, (iii) minimizing the size of the assertion.

The key challenge of defining a multi-objective fitness function is that these three objectives are competing with each other. Simply merging the objectives into the same fitness function is not an effective solution, as in MOOPs it is difficult to simultaneously reduce all competing objectives [31, 34, 41]. For an evolutionary algorithm, a possible strategy to improve a given program assertion might be either by first removing all false negatives (accepting more program behaviors, i.e., generalizing the assertion) or by removing false positives (accepting less program behaviors, i.e., specializing the assertion), or by an interleaving of these two strategies.

GAssert addresses this challenge with a co-evolutionary approach that evolves two populations in parallel with different fitness functions for each population. The fitness functions of the first and second population reward solutions with fewer false positives and false negatives, respectively, considering the remaining objectives only in tie cases. The two populations exchange their best individuals (population migration) on a regular basis, to supply both populations with good genetic material, useful to improve both the primary and secondary objectives. Moreover, GAssert presents novel crossover and mutation operators specifically designed for the oracle improvement problem.

We empirically evaluated GAssert on 34 methods from 7 Java code bases. We evaluated the ability of GAssert to improve an initial set of Daikon [9] generated assertions. The improved assertions eliminate all false positives present in the initial Daikon assertions, and reduce the false negatives by 40% (on average) with respect to the initial Daikon assertions. When executed with unseen tests and mutants, the GAssert assertions increase the mutation score by 34% (on average) with respect to the mutation score obtained with the initial assertions.

In summary, this paper makes the following contributions:

- We formulate the problem of automatically improving assertion oracles given a set of false positives and false negatives;
- We propose GAssert, the first technique to automatically improve assertion oracles;
- We evaluate GAssert on 34 methods from seven Java code bases, and show that GAssert outperforms both unguided-random and invariant-based approaches;
- We release our evaluation results (https://doi.org/10.5281/zenodo.3876638) and tool (https://doi.org/10.5281/zenodo.3877078) to facilitate future work in this area.

2 PROBLEM FORMULATION

This section provides the preliminaries for this work and formulates the problem of improving assertion oracles.

In this paper, $\mathcal{P}$ is an object-oriented program composed of a set of classes, each defining a set of methods and fields. Given a program point $\rho\rho$ of a method m in $\mathcal{P}$, $\mathcal{S}_{\rho\rho}$ denotes the set of all program states that can reach $\rho\rho$ when m is executed. A state $s \in \mathcal{S}_{\rho\rho}$ defines an assignment of values to memory locations that are accessible (visible) at the program point $\rho\rho$ (e.g., instance fields, method parameters and local variables). $\mathcal{S}_{\rho\rho}$ is partitioned into two disjoint sets: *correct* ($\mathcal{S}^+_{\rho\rho}$) and *incorrect* ($\mathcal{S}^-_{\rho\rho}$) program states. We say that a state is *correct* if it satisfies the intended program behaviour, *incorrect* otherwise. We drop the subscript $\rho\rho$ and use $\mathcal{S}$, $\mathcal{S}^+$ and $\mathcal{S}^-$ when $\rho\rho$ is clear from the context.

A program point $\rho\rho$ can be associated with an *assertion oracle* α, a quantifier-free first-order logic formula that predicates on variables and functions of Boolean or numerical types and returns a Boolean value (T or F). Let Σ denote the set of variables visible at the assertion point $\rho\rho$. Let $\mathcal{F}$ denote the set of Boolean and numerical operators that GAssert uses to synthesize assertions. The content of Σ depends on $\rho\rho$, while $\mathcal{F}$ is fixed for any $\rho\rho$. Table 1 shows the 17 functions in $\mathcal{F}$ grouped by operand and output type.

Assertion oracles aim to distinguish correct and incorrect executions. We consider assertions inserted into program $\mathcal{P}$, and not into its test cases. The difference is that assertions in $\mathcal{P}$ handle all possible test case executions, while assertions in the test cases check the correctness of a single test execution. More specifically, an *assertion oracle* α expresses a correctness property that is intended to be true at $\rho\rho$ in all correct executions (i.e., $\forall s^+ \in \mathcal{S}^+, \alpha[s^+] = T$) and false in all incorrect executions (i.e., $\forall s^- \in \mathcal{S}^-, \alpha[s^-] = F$), where $\alpha[s]$ denotes the evaluation of the Boolean expression α on state s. We call *perfect* oracle an assertion that satisfies such a condition.

Perfect oracles are difficult to design, and assertion oracles often fail to distinguish correct from incorrect executions, i.e., they have false positives and false negatives, which we call *oracle deficiencies*.

Definition 1. *A **false positive** of an assertion α at a program point $\rho\rho$ is a reachable program state where α is false, although such state is correct (according to the intended program behavior). More formally, it is a state $s^+ \in \mathcal{S}^+_{\rho\rho} : \alpha[s^+] = F$.*

Definition 2. *A **false negative** of an assertion α at a program point $\rho\rho$ is a reachable program state where α is true, although such state is incorrect (according to the intended program behavior). More formally, it is a state $s^- \in \mathcal{S}^-_{\rho\rho} : \alpha[s^-] = T$.*

In this paper, *we study the problem of automatically improving assertion oracles*, that is, given an assertion α and a set of oracle deficiencies, generating a new assertion α' with fewer deficiencies.

Table 1: Functions $\mathcal{F}$ Considered by GAssert

operand type	output type	functions
⟨ number, number ⟩	number	+, *, -, /, % (modulo)
⟨ number, number ⟩	boolean	==, <, >, ≤, ≥, ≠
⟨ boolean, boolean ⟩	boolean	AND, OR, XOR, EXOR, → (implies), == (equiv.)
⟨ boolean ⟩	boolean	NOT

Identifying oracle deficiencies by enumerating all correct and incorrect states is infeasible, because it requires to enumerate infinitely many executions [36]. Thus, we rely on a precise but incomplete **oracle assessor** $\mathcal{OA}$ that returns evidence of false positives and false negatives (if any) for a given assertion. We assume any $\mathcal{OA}$ to be *precise* (it reports only real oracle deficiencies), but possibly *incomplete* (it may miss oracle deficiencies) because it cannot enumerate all possible correct and incorrect executions.

An oracle assessor can be either a human or an automated technique. To enable full automation, we rely on the automated oracle assessor OASIs [20, 22]. Given an assertion α, OASIs leverages search-based test generation and mutation testing to report oracle deficiencies, if any can be found within the given time budget.

OASIs finds false positives of an assertion α by generating test cases that make α return false in the reached state. OASIs considers such states as false positives of α because it targets the *implemented* program behavior, which might differ from the *intended* one. As such, GAssert needs a manual validation of the improved assertions to ensure that they capture the intended program behavior.

OASIs finds false negatives of an assertion α by seeding artificial faults (mutations) into program $\mathcal{P}$ using mutation testing [13]. OASIs generates a test case and a mutation that produce a corrupted program state $s^- \in \mathcal{S}^-$ at the assertion point $\rho\rho$, where α does not reveal the fault, i.e., α returns true.

We now define the oracle improvement problem given an oracle assessor $\mathcal{OA}$. Let $\mathcal{A}$ denote the universe of possible Boolean expressions containing variables in Σ and functions in $\mathcal{F}$. To make $\mathcal{A}$ a finite set, we bound the size of assertions (i.e., the number of variables and functions in the assertions) to a maximum value (50 in our experiments). Let $\text{FP}(\alpha, \mathbb{S}^+)$ denote the number of false positives of α wrt a finite subset $\mathbb{S}^+$ of $\mathcal{S}^+$. That is, $\text{FP}(\alpha, \mathbb{S}^+)$ is the number of states $s^+ \in \mathbb{S}^+ \subseteq \mathcal{S}^+ : \alpha[s^+] = F$. Similarly, $\text{FN}(\alpha, \mathbb{S}^-)$ denotes the number of false negatives of α wrt a finite subset $\mathbb{S}^-$ of $\mathcal{S}^-$. That is, $\text{FN}(\alpha, \mathbb{S}^-)$ is the number of states $s^- \in \mathbb{S}^- \subseteq \mathcal{S}^- : \alpha[s^-] = T$.

Problem Definition 1. *Given an assertion α at a program point $\rho\rho$ in $\mathcal{P}$, given a set of false positives $\mathbb{S}^+ \subseteq \mathcal{S}^+$ and a set of false negatives $\mathbb{S}^- \subseteq \mathcal{S}^-$ reported by an oracle assessor $\mathcal{OA}$, and an overall time budget $\mathcal{B}$, the* ***oracle improvement*** *of α is the process of finding within $\mathcal{B}$ a new assertion $\alpha' \in \mathcal{A}$ such that $FP(\alpha', \mathbb{S}^+) = 0$ and either $FP(\alpha', \mathbb{S}^+) < FP(\alpha, \mathbb{S}^+)$ or $FN(\alpha', \mathbb{S}^-) < FN(\alpha, \mathbb{S}^-)$.*

In defining the oracle improvement, we give priority to false positive over false negative reduction, by requiring all false positives to disappear in the improved oracle α'. The rationale for this choice is that false negative reduction can be easily achieved with assertions that raise many false alarms. However, such assertions are troublesome for developers, as they trigger an expensive debugging process, in which the root of the assertion failures may likely be the assertion itself. Therefore, we privilege assertions with no false alarms (no false positives).

Ideally, the improved assertion oracle α' has zero oracle deficiencies wrt to $\mathbb{S}^+$ and $\mathbb{S}^-$ (i.e., $\text{FP}(\alpha', \mathbb{S}^+) = \text{FN}(\alpha', \mathbb{S}^-) = 0$). However, generating such assertions can be expensive and difficult, and may be infeasible within a reasonable time budget, as an oracle that detects all faults could be as complex as the method under test [20]. Therefore, we deem an oracle with zero false positives and the lowest number of false negatives sufficiently adequate in practice [22].

Algorithm 1: GAssert: Iterative Oracle Improvement Process

input : initial assertion α at progr. point $\rho\rho$ in $\mathcal{P}$, time-budget $\mathcal{B}$
output : improved assertion α'

```
1  function GAssert
2      P' ← INSTRUMENT-METHOD-AT-PROGRAM-POINT(ρρ, P)
3      ⟨S+, S−⟩ ← GET-INITIAL-CORRECT-AND-INCORRECT-STATES(P')
4      while time-budget B is not expired do
5          Σ ← GET-DICTIONARY-OF-VARIABLES(S+, S−)
6          α' ← ORACLE-IMPROVEMENT(α, S+, S−, Σ)
7          ⟨S+_NEW, S−_NEW⟩ ← ORACLE-ASSESSMENT(α') // OASIs [20]
8          if S+_NEW = ∅ ∧ S−_NEW = ∅ then
9              return α'
10         S+ ← S+ ∪ S+_NEW
11         S− ← S− ∪ S−_NEW
12         α ← α'
13     return α'
```

3 GASSERT

Algorithm 1 overviews the GAssert approach. GAssert's inputs are (i) an assertion oracle α, (ii) the program point $\rho\rho$ in $\mathcal{P}$ where α is placed, and (iii) a time budget $\mathcal{B}$. The output of GAssert is an improved assertion α'. GAssert improves assertion oracles with an iterative process. Before the first iteration, GAssert instruments $\mathcal{P}$ to capture program states at runtime (line 2 of Algorithm 1). It then produces an initial set of correct and incorrect states $\mathbb{S}^+$ and $\mathbb{S}^-$ by executing an initial test suite on the instrumented version $\mathcal{P}'$ and on its faulty versions (mutants), respectively (line 3). The while loop at lines 4–13 implements the iterative process. GAssert gets the dictionary of variables Σ from the states $\mathbb{S}^+$ and $\mathbb{S}^-$ (line 5), and invokes the oracle improvement of α (line 6). The ORACLE-IMPROVEMENT algorithm, which we discuss in Section 3.3, returns an improved assertion α' (line 6). If OASIs cannot find any oracle deficiencies of α', Algorithm 1 returns α', and the iterative process terminates (lines 8 and 9). Otherwise, GAssert adds the newly identified false positives and false negatives ($\mathbb{S}^+_{\text{NEW}}$ and $\mathbb{S}^-_{\text{NEW}}$) to $\mathbb{S}^+$ and $\mathbb{S}^-$ (lines 10 and 11), respectively. The improved assertion α' replaces the initial assertion α (line 12) and a new iteration starts.

3.1 Running Example

We now describe the GAssert oracle improvement process with a running example. Figure 1 shows a Java method that accepts two integers x and y as parameters $\vec{p}$, and returns the minimum between them. The figure also shows (i) the assertion point $\rho\rho$ (line 9), (ii) two instrumented method calls to collect the program states (lines 1 and 8), and (iii) two mutants M_1 and M_2 used to produce false negative program states (lines 6 and 4).

Table 2: Input and Output of the Oracle Assessor ($\mathcal{OA}$) of our Running Example

		False Positives (FP)		False Negatives (FN)		
iter.	**input assertion α**	**test**	**state**	**test**	**mutant**	**state**
0	$(min < x)$	t_1=min(x=3,y=5)	s_1^+={x=3,y=5,*min*=3}	t_2=min(x=9,y=7)	M_1	s_2^-={x=9,y=7,*min*=8}
1	$(min \leq x)$ AND $(min \leq y)$	∅	∅	t_3=min(x=3,y=7)	M_2	s_3^-={x=3,y=7,*min*=0}
2	$((min == x)$ OR $(min == y))$ AND $((min \leq x)$ AND $(min \leq y))$	∅	∅	∅	∅	∅

```
public static int min(int x, int y) {
1  serializer(x, y); // instrumentation
2  int min;
3  if (x <= y) {
4      min = x; // mutant M2: min = 0;
5  } else {
6      min = y; // mutant M1: min = y + 1;
7  }
8  serializer(x, y, min); // instrumentation
9  // program point pp of the assertion oracle
10 return min;
}
```

Figure 1: Java source code of the running example.

Table 2 illustrates how GAssert improves a trivially incomplete initial assertion ($min < x$) into a stronger assertion that intuitively captures the expected behavior of a "min" function: $((min == x)$ OR $(min == y))$ AND $((min \leq x)$ AND $(min \leq y))$. Column "*input assertion α*" shows the assertions that the oracle assessor ($\mathcal{OA}$) receives as input at each iteration. The first assertion ($min < x$) is provided to GAssert as an input, while the following two assertions are automatically generated by its evolutionary algorithm. The initial assertion can be manually generated or inferred with a tool. Column "*False Positives (FP)*" shows the false positive states with the test cases that produce them. On such states α fails while it should pass. Similarly, Column "*False Negatives (FN)*" shows the false negative states with the test cases and the mutants that produce them. On such states α passes but it should fail.

In the example, $\mathcal{OA}$ identifies both FPs and FNs for $\alpha : min < x$. Table 2 reports a test case t_1 that $\mathcal{OA}$ generates for α, and for which α incorrectly returns false. The execution of test cases t_1 produces the state s_1^+ that is a false positive for α (see Def. 1). The table also reports a sample test case t_2 and mutant M_1 that $\mathcal{OA}$ generates for α and for which α incorrectly returns true (α does not kill mutant M_1). The execution of test cases t_2 with mutant M_1 produces the state s_2^+ that is a false negative for α (see Def. 2).

At the first iteration, GAssert takes as input α, the false positive s_1^+ and the false negative s_2^- of α, and returns the improved assertion $\alpha' : (min \leq x)$ AND $(min \leq y)$. GAssert produces α' with an evolutionary algorithm that evolves populations of assertions towards an assertion with zero false positives and the lowest number of false negatives. The evolutionary algorithm explores the search space by (i) selecting pairs of assertions (parents) by means of fitness functions that reward solutions with fewer oracle deficiencies, (ii) creating new (and possibly fitter) offspring by exchanging genetic materials (portions of assertions) of the parents with crossover operators, and (iii) mutating the offspring (with a certain probability) using mutation operators.

We now exemplify how the evolutionary algorithm obtains $\alpha' : (min \leq x)$ AND $(min \leq y)$ during the first iteration. Let us assume that the algorithm selects two parents $\alpha_{p1} : min \leq x$ and $\alpha_{p2} : min \neq y$. The assertion α_{p1} reduces the number of false positives with respect to the initial assertion ($\text{FP}(\alpha_{p1}, \mathbb{S}^+) = 0$, where $\mathbb{S}^+ = \{s_1^+\}$), but it does not reduce the number of false negatives, because α_{p1} evaluates true under $\mathbb{S}^- = \{s_2^-\}$ ($\text{FN}(\alpha_{p1}, \mathbb{S}^-) = 1$). Conversely, the assertion α_{p2} reduces the number of false negatives ($\text{FN}(\alpha_{p2}, \mathbb{S}^-) = 0$), but it has the same number of false positives as α ($\text{FP}(\alpha_{p2}, \mathbb{S}^+) = \text{FP}(\alpha, \mathbb{S}^+) = 1$). The crossover operator *merge crossover* applied to α_{p1} and α_{p2} produces the offspring $\alpha_{o1} : (min \leq x)$ AND $(min \neq y)$ and $\alpha_{o2} : (min \leq x)$ OR $(min \neq y)$. If the mutation operators mutate α_{o1} into $(min \leq x)$ AND $(min \leq y)$, GAssert obtains an improved assertion with zero oracle deficiencies wrt $\mathbb{S}^+$ and $\mathbb{S}^-$, and the first iteration terminates.

At the second iteration, $\mathcal{OA}$ takes in input $\alpha : (min \leq x)$ AND $(min \leq y)$ to find its oracle deficiencies (if any). For this assertion, $\mathcal{OA}$ does not find false positives, but it reports a false negative: executing test t_3 with mutant M_2 leads to the state s_3^-, which is a false negative for α ($\text{FN}(\alpha, \mathbb{S}^-) = 1$, where $\mathbb{S}^- = \{s_2^-, s_3^-\}$). In fact, α returns true under s_3^-, while it should return false.

Given the assertion $\alpha : (min \leq x)$ AND $(min \leq y)$, the correct states $\mathbb{S}^+ = \{s_1^+\}$ and the incorrect states $\mathbb{S}^- = \{s_2^-, s_3^-\}$, the evolutionary algorithm returns the improved assertion $\alpha' : ((min == x)$ OR $(min == y))$ AND $((min \leq x)$ AND $(min \leq y))$. This assertion does not have oracle deficiencies wrt $\mathbb{S}^+$ and $\mathbb{S}^-$ (i.e., $\text{FP}(\alpha', \mathbb{S}^+) = \text{FN}(\alpha', \mathbb{S}^-) = 0$). As $\mathcal{OA}$ does not find oracle deficiencies for α', the GAssert improvement process terminates.

The following two subsections describe in detail how GAssert serializes program states and how it improves assertion oracles.

3.2 Program State Serialization

A program state $s = \{v_1, \cdots, v_n\}$ is a set of variables that are in memory at a certain execution point. Each variable v_i has a type $type(v_i)$, an identifier $id(v_i)$ and a value $value(v_i)$. Deciding which variables compose a program state is a key design choice. It defines both the expressiveness of the assertions that GAssert can produce (the dictionary Σ) and the size of the search space ($\mathcal{A}$). GAssert should consider variables that capture useful properties of the method under test, and ignore irrelevant ones. Indeed, considering too many variables unnecessarily increases the search space, which hardens the problem of finding oracle improvements.

Given a method $m(\vec{p})$ with formal parameters $\vec{p}$, GAssert constructs the program state s at $\rho\rho$ considering as variables all parameters p_i of $\vec{p}$ and all the local variables created in m that are visible at $\rho\rho$. Note that when $m(\vec{p})$ is a non-static method, the object receiver of m (`this` in Java) is m's first parameter p_0. GAssert captures the values of the parameters both at the beginning of the method (adding the prefix "*old*" to the variable identifiers) and immediately before $\rho\rho$. By considering "*old*" values, GAssert can generate assertions that predicate on method preconditions [9].

When the considered variable has a primitive type, GAssert simply adds its runtime value to the program state (rounding floats with a fixed precision) using the variable name as identifier. However, variables of object-oriented programs can have both primitive and non-primitive (object) types, introducing the problem of obtaining primitive values from objects. Given a non-primitive variable v_i, there are two well-established approaches to obtain primitive values: object serialization and observer abstraction. *Object serialization* [42] captures the values of all primitive-type object fields that are recursively reachable from v_i. *Observers abstraction* [2] captures the return values of observer methods invoked with v_i as the object receiver. Observer methods are side-effect free methods that are declared in v_i's class and return primitive values. Such values often characterize important properties of objects [1, 2].

Both approaches have advantages and disadvantages. Object serialization can lead to many variables, which unnecessarily increase the search space. Indeed, many recursively obtained primitive variables often refer to implementation details that do not capture interesting properties of objects. Observers abstraction is inherently incomplete because the available observer methods might not capture all the relevant aspects of the analyzed objects [2].

Hybrid State Serialization. To address the issue, GAssert opts for a hybrid solution that combines both approaches. We rely on observer methods for all non-primitive variables considered by GAssert. In addition, we use the object serialization approach only for the object receiver (`this`) of the method under test m, capturing the values of all primitive fields of `this`. For non-primitive fields of `this`, we do not serialize their recursively reachable primitive fields, but again we use the observers abstraction approach. The rationale is that the primitive fields of m's object receiver are more likely to capture important aspects of the behavior of m than recursively reachable primitive fields or other method parameters. We now describe in detail our hybrid approach.

Let v_i be a non-primitive variable considered for constructing the program state s. GAssert finds the observer methods $\{f_1, f_2, \ldots, f_n\}$ of the class C of v_i by using a static analyzer that scans the bytecode instructions of the public methods in C. The analyzer marks a method f_j as *observer* if (i) f_j returns a number or a Boolean, and (ii) f_j cannot directly or indirectly execute `putfield` or `putstatic` bytecode instructions (m is side-effect free), and (iii) f_j does not have parameters (besides the object receiver). When collecting state s at runtime, for each observer method f_j with return type τ_j, GAssert adds to state s a variable with identifier "$id(v_i).f_j$", type τ_j and value the result of the invocation of $v_i.f_j$.

For non-primitive variables of type array, string or Java collection (objects that extend `java.util.Collection`) GAssert considers a smaller set of observer methods that capture the most important properties of such object types. GAssert adds $v_i.size$ ($v_i.length$ for arrays and String) and $v_i.isEmpty$ to the state s.

If v_i is the object receiver (i.e., $id(v_i)$ = `this`), GAssert serializes it by adding variable `this`.$field_j$ to the state s, for each primitive-type field $field_j$ of `this`. It then applies the observer methods approach described above to each non-primitive fields of `this`.

Collecting States. Function INSTRUMENT-METHOD-AT-PROGRAM-POINT instruments the method m that contains $\rho\rho$ by adding two method calls (Algorithm 1, line 2). One at the beginning of the method (to get the "*old*" values) and the other immediately before $\rho\rho$. When a test execution reaches the instrumented method calls, GAssert performs the state serialization described above. Every time GAssert executes a new test, it stores the observed states so that the fitness functions can compute the number of FP and FN without requiring expensive program re-executions.

Initial Program States. Function GET-INITIAL-CORRECT-AND-INCORRECT-STATES (line 3 Algorithm 1) generates a set of initial correct ($\mathbb{S}^+$) and incorrect ($\mathbb{S}^-$) program states by executing an initial test suite on both the instrumented program $\mathcal{P}'$ and its faulty versions. The rationale of considering these initial states (as opposed to immediately relying on the oracle assessor $\mathcal{OA}$) is to minimize the number of iterations of the while loop (line 4 Algorithm 1). In this way, GAssert avoids invoking $\mathcal{OA}$ to detect obvious oracle deficiencies, and rather lets $\mathcal{OA}$ focus on hard-to-find ones.

Post-processing the States. Function GET-INITIAL-CORRECT-AND-INCORRECT-STATES post-processes the states with two scans. The first scan removes redundant states from $\mathbb{S}^+$, so that $\nexists s_1, s_2 \in \mathbb{S}^+$ such that s_1 and s_2 are *equivalent* ($s_1 \equiv s_2$), i.e., all corresponding variables have identical values: $\forall v_1 \in s_1, \forall v_2 \in s_2$, if $id(v_1) = id(v_2)$ then $value(v_1) = value(v_2)$. The second scan checks that each state in $\mathbb{S}^-$ is indeed incorrect, i.e., the seeded fault (the mutant) has successfully corrupted the program state. For each incorrect state $s^- \in \mathbb{S}^-$, GAssert retrieves the correct state $s^+ \in \mathbb{S}^+$, obtained when executing the same test that produced s^- on the original version of the program (without the seeded fault). If $s^- \equiv s^+$, GAssert found a *likely equivalent state* and removes s^- from $\mathbb{S}^-$. We call them *likely* because our collected states encode only a fragment of the actual program state.

Dictionary of Variables. Function GET-DICTIONARY-OF-VARIABLES (line 5 Algorithm 2) builds the dictionary of variables Σ that function ORACLE-IMPROVEMENT uses to create new assertions. The function picks an arbitrary state s in either $\mathbb{S}^+$ or $\mathbb{S}^-$ (by construction all states have the same variables), and adds all the variables in s to Σ.

3.3 Oracle Improvement

A major challenge to automatically improve assertion oracles is the huge search space of candidate solutions ($\mathcal{A}$ in Section 2), which grows exponentially with the number of variables and functions.

GAssert addresses this challenge with Genetic Programming (GP) [3, 45]. We formulate the oracle improvement problem as a multi-objective optimization problem (MOOP) [31, 34, 41] with three competing objectives: (i) minimize the number of false positives (FP); (ii) minimize the number of false negatives (FN); (iii) minimize the size of the assertion, that is, the number of variables and functions in it. The latter objective helps to improve the quality of assertions, as long assertions are often difficult to understand.

Classic multi-objective evolutionary approaches, for instance NSGA-II [8, 43], rely on *Pareto optimality* [18, 39, 41] to produce solutions that offer the best trade-off between competing objectives [41]. However, in our case not all assertions with an optimal trade-off between FPs and FNs are acceptable solutions. As discussed in Section 2, we aim to obtain assertions with zero FPs and the lowest number of FNs. On the other hand, primarily focusing on reducing FPs may be inadequate, as there may not be enough *evolution pressure* [45] to reduce the FNs at the same time.

Hence, we propose a **co-evolutionary approach** that evolves in parallel two distinct populations of assertions ($Popul^{FP}$ and $Popul^{FN}$) with two competing objectives: reduce the false positives (fitness function ϕ_{FP}) and reduce the false negatives (fitness function ϕ_{FN}). These populations periodically exchange their best individuals (population migration) to add promising genetic material in both populations. Eventually, $Popul^{FP}$ will more likely produce assertions with zero FPs and fewer FNs. In fact, the migration of best individuals adds in $Popul^{FP}$ assertions with a decreasing number of FNs.

Fitness Functions. Both ϕ_{FP} and ϕ_{FN} are multi-objective fitness functions. The former gives priority to reducing false positives, while the latter to reducing false negatives. Both functions consider the remaining objectives only in tie cases. In multi-objective optimization, the fitness of a solution is often defined by the concept of *dominance* ($\prec$) [8]. While the standard definition of dominance gives the same importance to all objectives, we need an unbalanced definition towards FPs and FNs, which we define as follows:

Definition 3. ***FP-fitness (ϕ_{FP}).*** *Given two assertions α_1 and α_2 and two sets of correct $\mathbb{S}^+$ and incorrect $\mathbb{S}^-$ states, α_1 **dominates**$_{\text{FP}}$ α_2 ($\alpha_1 \prec_{FP} \alpha_2$) if any of the following conditions is satisfied:*

- $FP(\alpha_1, \mathbb{S}^+) < FP(\alpha_2, \mathbb{S}^+)$
- $FP(\alpha_1, \mathbb{S}^+) = FP(\alpha_2, \mathbb{S}^+) \ \wedge \ FN(\alpha_1, \mathbb{S}^-) < FN(\alpha_2, \mathbb{S}^-)$
- $FP(\alpha_1, \mathbb{S}^+) = FP(\alpha_2, \mathbb{S}^+) \ \wedge \ FN(\alpha_1, \mathbb{S}^-) = FN(\alpha_2, \mathbb{S}^-)$ $\wedge \ size(\alpha_1) < size(\alpha_2)$

Definition 4. ***FN-fitness (ϕ_{FN}).*** *Given two assertions α_1 and α_2 and two sets of correct $\mathbb{S}^+$ and incorrect $\mathbb{S}^-$ states, α_1 **dominates**$_{\text{FN}}$ α_2 ($\alpha_1 \prec_{FN} \alpha_2$) if any of the following conditions is satisfied:*

- $FN(\alpha_1, \mathbb{S}^-) < FN(\alpha_2, \mathbb{S}^-)$
- $FN(\alpha_1, \mathbb{S}^-) = FN(\alpha_2, \mathbb{S}^-) \ \wedge \ FP(\alpha_1, \mathbb{S}^+) < FP(\alpha_2, \mathbb{S}^+)$
- $FN(\alpha_1, \mathbb{S}^-) = FN(\alpha_2, \mathbb{S}^-) \ \wedge \ FP(\alpha_1, \mathbb{S}^+) = FP(\alpha_2, \mathbb{S}^+)$ $\wedge \ size(\alpha_1) < size(\alpha_2)$

In tie cases, $FP(\alpha_1, \mathbb{S}^+) = FP(\alpha_2, \mathbb{S}^+)$ and $FN(\alpha_1, \mathbb{S}^-) = FN(\alpha_2, \mathbb{S}^-)$, both functions favor smaller assertions. If neither $\alpha_1 \prec \alpha_2$ nor $\alpha_2 \prec \alpha_1$, the choice between α_1 and α_2 is random. We now describe the details of our co-evolutionary algorithm (Algorithm 2).

Building the Initial Populations. Both populations $Popul^{FP}$ and $Popul^{FN}$ contain N assertions each. We represent an assertion $\alpha \in Popul$ as a rooted binary tree [7], where leaf nodes are variables or constants (terminals) and inner nodes are functions. Each node has a type, either Boolean or numerical. The type of leave nodes is the type of the associated variable, the type of the inner nodes is the type of the function outputs. We define the size of an assertion α, $size(\alpha)$, as the number of nodes in its tree representation.

Function GET-INITIAL-POPULATION at lines 2 and 3 of Algorithm 2 initializes the two populations, $Popul^{FP}$ and $Popul^{FN}$, respectively, in the same way. Half of the initial population consists of randomly-generated assertions (to guarantee genetic diversity), the other half of assertions is obtained by randomly mutating the input assertion α (to have "good" genetic material for evolution). Intuitively, an improved assertion could include fragments similar to the input assertion, thus initializing the populations with variants of α increases the chances of introducing "good" genetic material.

GAssert produces the first half of individuals with a **tree factory** operator that takes a type τ (either number or Boolean) and a depth d, and returns a randomly-generated assertion with root of type τ and depth of the tree d. Because the root of an assertion must be of Boolean type, GAssert always sets τ to Boolean, and invokes *tree factory* $N/2$ times with random values of d.

Algorithm 2: Evolutionary Improvement of Assertion Oracles

input : correct $\mathbb{S}^+$ and incorrect $\mathbb{S}^-$ states, assertion oracle α
output: improved assertion oracle α'

```
1  function ORACLE-IMPROVEMENT
2      Popul^FP ← GET-INITIAL-POPULATION(α, Σ)
3      Popul^FN ← GET-INITIAL-POPULATION(α, Σ)
4      for gen from 1 to max-number-of-gen do
5          if ∃a' ∈ {Popul^FP ∪ Popul^FN} : FP(a', S+) = FN(a', S-) = 0 AND
6          gen ≤ min-number-of-gen then
7              return a'
8          do in parallel
9              Popul^FP ← SELECT-AND-REPRODUCE(Popul^FP, φ_FP, Σ, S+, S-)
10             Popul^FN ← SELECT-AND-REPRODUCE(Popul^FN, φ_FN, Σ, S+, S-)
11         if gen % frequency-migration = 0 then
12             do in parallel
13                 Popul^FP ← MIGRATE(Popul^FN, φ_FP, φ_FN)
14                 Popul^FN ← MIGRATE(Popul^FP, φ_FN, φ_FP)
15     return a' with zero FP and the lowest number of FN
```

input : population *Popul*, fitness function ϕ, dictionary of variables Σ, correct $\mathbb{S}^+$ and incorrect $\mathbb{S}^-$ states, generation count *gen*
output: new population $Popul_{\text{NEW}}$

```
16 function SELECT-AND-REPRODUCE
17     Popul ← COMPUTE-FITNESS(Popul, S+, S-)
18     Popul_NEW ← ∅
19     if gen % frequency-of-elitism = 0 then
20         Popul_NEW ← GET-BEST-INDIVIDUALS(φ)
21     while Popul_NEW is not full do
22         ⟨a_p1, a_p2⟩ ← SELECT-PARENTS(Popul, φ)
23         ⟨a_o1, a_o2⟩ ← CROSSOVER-AND-MUTATION(a_p1, a_p2, Σ)
24         add ⟨a_o1, a_o2⟩ to Popul_NEW
25     return Popul_NEW
```

Tree Mutations. To obtain the second half of individuals, GAssert relies on two classic tree-based mutation operators:

Node Mutation changes a single node in the tree [5]. It takes as input an assertion α and one of its nodes n, and returns an assertion α_1 obtained by replacing the node n in α_1 with a new node with the same type of n (chosen randomly).

Subtree Mutation replaces a subtree in the tree [5]. It takes as input an assertion α and one of its nodes n, and returns a new assertion a_1 obtained by substituting the subtree rooted at n with another subtree. Such a subtree is generated by the *tree factory* operator with the type of n as τ and a random number as d.

Stopping Criterion. Algorithm 2 evolves the two populations in parallel until either $Popul^{FP}$ or $Popul^{FN}$ contains a perfect assertion α' with respect to the correct and incorrect states in input (line 6 of Algorithm 2). If GAssert finds the perfect assertion before a maximum number of generations, it continues the evolution process to see if it can find perfect assertions of smaller size. Algorithm 2 prematurely terminates when the overall time-budget $\mathcal{B}$ expires or when it reaches a maximum number of generations. In both cases, GAssert returns the best generated assertion, the one with zero false positives and the lowest number of false negatives, that is, α' s.t. $\nexists \alpha \in \{Popul^{FP} \cup Popul^{FN}\} : \alpha \prec_{FP} \alpha'$ (line 15 Algorithm 2).

Lines 9 and 10 of Algorithm 2 evolve in parallel the two populations by invoking function SELECT-AND-REPRODUCE (lines 17-25). The function implements the classic evolutionary approach [45], which works in three consecutive steps: selection, crossover and mutation. GAssert introduces novel selection and crossover operators that are specific for the automatic oracle improvement problem.

Fitness Computation. GAssert initializes the selection process by computing the number of false positives $FP(\alpha, \mathbb{S}^+)$ and false negatives $FN(\alpha, \mathbb{S}^-)$ for each $\alpha \in Popul$ (function COMPUTE-FITNESS line 17 Algorithm 2). Both fitness functions need this information to compute the dominance relation. GAssert optimizes the fitness computation by (i) loading the $\mathbb{S}^+$ and $\mathbb{S}^-$ states in the primary memory to avoid costly re-executions of the program, (ii) parallelizing the computation, (iii) caching the results to avoid recomputing them upon encountering the same assertion multiple times.

Function SELECT-AND-REPRODUCE initializes the new population $Popul_{\text{NEW}}$ with the empty set (line 18 of Algorithm 2) performing the elitism if *gen % frequency-migration = 0*. It then proceeds with parent selection, parent crossover and offspring mutation, adding the resulting offspring to $Popul_{\text{NEW}}$ until $Popul_{\text{NEW}}$ reaches size N.

Parent Selection. Function SELECT-PARENTS selects two parents a_{p1} and a_{p2} from *Popul* (line 22 in Algorithm 2). GAssert implements two different selection criteria, tournament and best-match selection, and chooses between them with a given probability.

Tournament Selection [28] is a classic GP selection criterion [45]. It runs "tournaments" among K randomly-chosen individuals and selects the winner of each tournament (the one with the highest fitness) [28]. As GAssert needs two parents, it plays two tournaments to obtain α_{p1} and α_{p2}. We choose $K = 2$ (the most commonly used value [26]) as it mitigates the *local optima problem* [45].

Best-match Selection is a new criterion presented in this paper, which is specific to the oracle improvement problem. The criterion exploits semantic information about the correct and incorrect states that each assertion *covers*. Let $cov^+(\alpha,\mathbb{S}^+)$ denote the subset of $\mathbb{S}^+$ on which α evaluates to true, i.e., $cov^+(\alpha,\mathbb{S}^+) = \{s^+ \in \mathbb{S}^+ : \alpha[s^+] = T\} \subseteq \mathbb{S}^+$. Let $cov^-(\alpha,\mathbb{S}^-)$ denote the subset of $\mathbb{S}^-$ on which α evaluates to false, i.e., $cov^-(\alpha,\mathbb{S}^-) = \{s^- \in \mathbb{S}^- : \alpha[s^-] = F\} \subseteq \mathbb{S}^-$. The best-match criterion selects the first parent α_{p1} randomly from *Popul*. If *Popul* is $Popul^{FP}$, the best-match selection criterion gets the set of all assertions $\alpha_1 \in Popul^{FP}$ such that $\{cov^+(\alpha_1,\mathbb{S}^+) \backslash cov^+(\alpha, \mathbb{S}^+)\} \neq \varnothing$. For each assertion α_1 in the set, the best-match selection criterion considers the cardinality of $\{cov^+(\alpha_1,\mathbb{S}^+) \backslash cov^+(\alpha,\mathbb{S}^+)\}$ as the *weight* of α_1. It then selects the second parent α_{p2} from the set using a *weighted random selection*, where assertions with a higher weight are more likely to be selected. Symmetrically, if *Popul* is $Popul^{FN}$, the best-match criterion considers cov^- instead of cov^+. Intuitively, the criterion increases the chances of crossover between two complementary individuals that are likely to yield a fitter offspring.

Crossover. Function CROSSOVER-AND-MUTATION exchanges genetic material between two parents α_{p1} and α_{p2}, producing two offspring α_{o1} and α_{o2}, which GAssert mutates (with a given probability) with the mutation operators used to initialize the two populations. GAssert implements two crossover operators, subtree and merging crossover, and chooses between them with a given probability.

Subtree Crossover [24] is the canonical tree-based crossover. Given two parents, it selects a crossover point in each parent, and creates the offspring α_{o1} and α_{o2} by swapping the subtrees rooted at each point in the corresponding tree [24].

Merging Crossover is an operator that we specifically defined for the oracle improvement problem. Given two parents α_{p1} and α_{p2}, it selects two Boolean subtrees, α_1 from α_{p1} and α_2 from α_{p2}, and creates the offspring $\alpha_{o1} : (\alpha_1\ AND\ \alpha_2)$ and $\alpha_{o2} : (\alpha_1\ OR\ \alpha_2)$. This operator works well in synergy with our best-match criterion, since merging two subtrees with OR and AND functions combines their semantics without disrupting them.

Node Selectors. A key design choice is the criterion to select the nodes of the tree α for crossover and mutation. We implemented two different selection criteria: (i) *Random* that randomly selects a node in α, (ii) *Mutation-based* that selects a node in α such that the subtree rooted on this node contains at least a variable v_i with the following property: $\exists s^- \in \mathbb{S}^-$ in which the value of v_i differs from the value in the corresponding state $s^+ \in \mathbb{S}^+$ obtained when executing on the original program the same test that yielded s^-. As such, v_i can recognize s^- as a false negative of α. A such, a new assertion that predicates on v_i could solve such a false negative.

Migration. GAssert periodically exchanges the M best individuals between the two populations, where M is a hyper parameter of the algorithm (see lines 11-14 of Algorithm 2). When selecting the best individuals GAssert considers both fitness functions so that both populations can benefit from assertions that have either the lowest number of false positives or false negatives.

4 EVALUATION

To experimentally evaluate our approach, we developed a prototype implementation of GAssert for Java classes. We conducted a series of experiments to answer three research questions:

- **RQ1** *Is GAssert effective at improving assertion oracles?*
- **RQ2** *How does GAssert compare with random (unguided) and invariant-based oracle improvement?*
- **RQ3** *How does GAssert compare with human oracle improvement?*

RQ1 evaluates the effectiveness of our evolutionary algorithm. RQ2 checks whether our fitness functions provide useful guidance to improve assertions. Towards this goal, we compare GAssert with a version of GAssert (Random) where the guidance provided by our fitness functions is replaced by a random choice. As a further baseline, RQ2 compares GAssert with an oracle improvement process (Inv-based) that relies on the invariant generator Daikon [9] to improve assertions. RQ3 compares our automated approach with oracle improvement performed by humans.

4.1 Subjects

We conducted our experiments on 34 Java methods. We took four methods from the *SimpleExamples* (SE) class, used by the recent Jahangirova et al.'s oracle improvement study [22]. Ten methods from the Daikon subjects *StackAr* (SA) and *QueueAr* (QA), often used to evaluate Java invariant generators. We selected the remaining 20 methods from four popular Java libraries: Apache Commons Math (CM), Apache Commons Lang (CL), Google Guava (GG) and JTS Topology Suite (TS). From each library we randomly selected

five methods with the following characteristics: (i) contain at least five lines of code, (ii) produce a return value, (iii) are not recursive, (iv) do not write to files and do not use reflection (as the outcome of such operations cannot be captured by our assertion oracles). For each of the 34 methods, we selected as the program point $\rho\rho$ of the assertion the last exit point of the method.

4.2 Evaluation Setup

To run GAssert we need an initial test suite and perform mutation analysis to get an initial set of incorrect states, in addition to the correct states obtained by running the initial test cases on the original program. We also need some initial assertions to be improved. We evaluate the improved assertions with the number of false positives and false negatives on the initial test cases and mutations. To avoid circularity in the evaluation we collected a new sets of validation test cases and mutations. The number of false positives and the mutation score obtained on the latter provide an external assessment of effectiveness.

Initial Test Cases and Mutations. We obtained the initial correct states ($\mathbb{S}_0^+$) by executing an initial test suite ($\mathcal{T}_0$) on the instrumented version of $\mathcal{P}$. We generated $\mathcal{T}_0$ by running EvoSuite [12, 13] (v. 1.0.6) with the branch coverage criterion and a time budget of one minute [12]. We performed ten runs with different random seeds to collect a diverse and large set of initial test cases.

We obtained the initial incorrect states ($\mathbb{S}_0^-$) by executing the initial test suite ($\mathcal{T}_0$) on a set of initial mutations ($\mathcal{M}_0$) of the instrumented version of $\mathcal{P}$. We obtained such mutations by running Major [23] (v. 1.3.4) enabling all types of supported mutants.

Columns "$|\mathbb{S}_0^+|$" and "$|\mathbb{S}_0^-|$"of Table 4 show the cardinality of the initial states. Note that for some subjects $|\mathbb{S}_0^-| < |\mathbb{S}_0^+|$, which is counterintuitive. This is because GAssert removes redundant states from both $\mathbb{S}_0^+$ and $\mathbb{S}_0^-$, and likely equivalent states from $\mathbb{S}_0^-$.

Initial Assertion Oracles. We obtained an initial assertion for our subjects by running the dynamic invariant generator Daikon [9] (v. 5.7.2) with the initial test suite in input. We chose Daikon because is a fully-fledged tool and is the de-facto invariant generator for Java methods [9]. Because also Daikon accepts in input a set of observer methods, we ran Daikon with the same observer methods that GAssert used to serialize program states.

Daikon generates invariants considering all possible exit points of a method (e.g., returns and exception throw statements). However, our oracle improvement process focuses on a single program point $\rho\rho$. To ensure that Daikon generates invariants that consider only the exit point at $\rho\rho$, we automatically remove all the initial test cases that do not reach $\rho\rho$ (i.e., do not produce program states).

Daikon outputs invariants as a series of precondition α_1, α_2, $\ldots\alpha_n$ and postcondition assertions $\beta_1, \beta_2, \ldots\beta_m$ [9]. GAssert converts them into a single (complete) Java assertion in the form of $(\alpha_1 \text{ AND } \alpha_2, \ldots \text{ AND } \alpha_n) \rightarrow (\beta_1 \text{ AND } \beta_2, \ldots \text{ AND } \beta_n)$.

GAssert initializes half of the populations by adding the complete assertion, all single α and β, and random mutations of each of these assertions. GAssert initializes the other half of the populations with randomly generated assertions.

Validation Test Cases and Mutations. To evaluate if the improved assertions generalize well with unseen correct and incorrect states, we generated new test cases ($\mathcal{T}_v$) and mutations ($\mathcal{M}_v$). We obtained such validation sets using the tools Randoop (v. 4.2.0) and PIT (v. 1.4.0), respectively. These tools are different from the ones that provide test cases and mutations to the oracle improvement process (EvoSuite, Major and OASIs). Different tools are expected to obtain different test cases and mutations.

For each subject, we ran Randoop ten times with different random seeds using at least 100 test cases or three minutes as stopping criterion. We ran PIT enabling all types of supported mutants. Columns "$|\mathcal{T}_v|$" and "$|\mathcal{M}_v|$" of Table 4 show the cardinality of the validation test cases and mutations, respectively.

Table 3: GAssert Configuration Parameters Values

Parameter Description	Value	Parameter Description	Value
bound on the size of the assertions	50	prob. of crossover	90%
size of each of the populations (N)	1,000	prob. of mutation	20%
minimum number of generations	100	prob. of tournament parent selection	50%
maximum number of generations	10,000	prob. of best-match parent selection	50%
frequency of elitism (every X gen)	1	prob. of merging crossover	50%
frequency of migration (every X gen)	100	prob. of random crossover	50%
number of assertions for elitism	10	prob. of mutate-state-diff node selector	30%
number of assertions to migrate (M)	160	prob. of random node selector	70%

Quality Metrics for Assertions. We evaluate an improved assertion α' by comparing the number of FPs and FNs of α' with the number of FPs and FNs of the initial assertion α wrt the initial and validation sets of test cases ($\mathcal{T}$) and mutations ($\mathcal{M}$).

Before evaluating the assertions, we removed from all the test cases the test oracle assertions that EvoSuite and Randoop generated. We then inserted the assertion under evaluation (either α or α') into the method under analysis at the specified $\rho\rho$.

To evaluate an assertion with the initial sets, we executed $\mathcal{T}_0$ and count the number of failing tests, which represents the number of false positives FP($\alpha, \mathbb{S}_0^+$), FP_0 in short. If FP_0 is zero, we compute FN($\alpha, \mathbb{S}_0^-$), FN_0 in short, by running mutation testing with mutations $\mathcal{M}_0$ and test cases $\mathcal{T}_0$. If FP_0 is greater than zero, we cannot run mutation testing because we need a green test suite. In such a case, if the evaluated assertion has the form *assert(α_1 AND α_2 AND α_3)*, we consider each of the smaller assertions *assert(α_1)*, *assert(α_2)* and *assert(α_3)* and remove those that have false positives. We concatenate the remaining smaller conditions with ANDs and perform mutation testing with $\mathcal{M}_0$ and $\mathcal{T}_0$ for this reduced assertion at $\rho\rho$. If all smaller assertions have false positives then we report FN_0 for the assertion oracle *assert(true)*.

We follow the same procedure to evaluate an assertion with the validation test cases ($\mathcal{T}_v$) and mutations ($\mathcal{M}_v$). We use FP_v to denote the number of false positives of an assertion wrt $\mathcal{T}_v$. While Major returns the source code of each mutation, PIT does not. Thus, we cannot compute the number of false negatives wrt $\mathcal{T}_v$ and $\mathcal{M}_v$ but only the mutation score (denoted by $\mathcal{M}_v$%).

Configuration. Table 3 shows the GAssert configuration parameters values used in our experiments. We selected these values after some trial runs following popular GP guidelines [3, 45]. We ran GAssert with an overall time budget $\mathcal{B}$ of 90 minutes. To ensure that GAssert will leverage the feedback of OASIs, we set an internal time budget of the oracle improvement process to 30 minutes. As such, GAssert must receive the feedback of OASIs at least two times. To cope with the stochastic nature of GP, we ran GAssert ten times with the same input assertion and initial correct and incorrect states. We implemented GAssert to be pseudo-deterministic given

Table 4: Evaluation Results for RQ1 and RQ2

subj.	evaluation sets				RQ1											RQ2										
					initial assertion α					GAssert improved α' (median)						Random improved α' (median)						Inv-based improved α' (median)					
ID	$\mathbb{S}_0^+$	$\mathbb{S}_0^-$	$\mathcal{T}_v$	$\mathcal{M}_v$	FP_0	FN_0	FP_v	$\mathcal{M}_v$%	size	# iter.	FP_0	FN_0	FP_v	$\mathcal{M}_v$%	size	# iter.	FP_0	FN_0	FP_v	$\mathcal{M}_v$%	size	# iter.	FP_0	FN_0	FP_v	$\mathcal{M}_v$%	size
SE1	36	46	1,010	4	0	10	0	75%	7	3	0	0	0	75%	15	3	0	12	0	75%	11	1	0	10	0	75%	7
SE2	20	250	1,062	8	0	0	0	100%	17	3	0	0	0	100%	17	4	0	10	0	88%	9	1	0	0	0	100%	17
SE3	41	250	1,199	15	0	32	0	33%	23	3	0	32	0	33%	3	3	0	32	0	33%	3	1	0	32	0	33%	23
SE4	10	20	54	7	0	10	0	0%	1	4	0	0	0	57%	23	4	0	0	0	0%	10	-	-	-	-	-	-
SA1	10	30	1,026	10	0	10	0	60%	43	1	0	0	0	50%	5	4	0	0	0	50%	5	1	0	10	0	60%	43
SA2	68	57	1,026	8	0	12	0	100%	51	3	0	48	0	100%	5	3	0	48	0	100%	5	68	0	12	0	100%	51
SA3	22	20	1,062	8	0	20	0	88%	9	3	0	20	0	50%	5	3	0	20	0	50%	5	72	0	20	0	88%	9
SA4	10	70	1,062	12	0	20	0	67%	37	9	0	0	0	67%	7	6	0	0	0	67%	7	88	0	20	0	67%	37
SA5	13	101	1,026	8	0	13	61	100%	135	3	0	11	0	100%	8	3	0	11	0	100%	7	66	12	13	61	100%	135
QA1	10	90	1,004	9	0	0	89	67%	5	5	0	0	0	89%	15	4	0	3	0	**89%**	13	68	0	0	89	67%	5
QA2	59	184	1,004	16	0	121	0	88%	127	3	0	56	0	100%	13	3	0	66	0	88%	7	1	0	121	0	88%	127
QA3	42	162	1,004	19	0	31	0	74%	87	3	0	30	0	84%	38	3	0	76	0	63%	12	1	0	31	0	74%	87
QA4	21	52	835	19	0	0	0	0%	147	4	0	0	0	84%	25	4	0	20	0	68%	12	1	0	0	0	0%	147
QA5	21	20	1,004	4	0	20	0	100%	45	3	0	20	0	100%	5	3	0	20	0	100%	6	68	0	20	0	100%	45
CM1	34	53	1,900	20	0	45	0	10%	1	3	0	0	0	25%	5	3	0	0	0	25%	5	-	-	-	-	-	-
CM2	58	359	872	44	0	359	0	5%	1	3	0	347	0	13%	10	3	0	341	0	23%	11	-	-	-	-	-	-
CM3	41	202	741	28	0	75	0	11%	1	3	0	0	0	18%	13	3	0	69	0	11%	7	-	-	-	-	-	-
CM4	29	467	860	30	0	459	0	0%	83	3	0	173	0	60%	49	3	0	440	0	13%	7	1	0	459	0	40%	83
CM5	59	415	1,881	27	0	277	0	7%	1	3	0	42	0	63%	9	1	0	42	0	44%	3	-	-	-	-	-	-
CL1	21	86	380	9	0	36	0	0%	1	3	0	30	0	0%	7	3	0	36	0	0%	7	-	-	-	-	-	-
CL2	30	128	114	9	0	128	0	0%	11	3	0	128	0	0%	4	3	0	128	0	0%	4	1	0	128	0	0%	11
CL3	19	542	1,736	23	0	70	0	39%	5	3	0	46	0	48%	3	3	0	47	0	39%	3	2	0	70	0	39%	5
CL4	220	1,502	114	81	0	1,282	0	10%	3	3	0	1,282	0	10%	3	3	0	1,282	0	10%	3	1	0	1,282	0	10%	3
CL5	35	306	1,881	36	2	-	0	47%	5	3	0	207	0	**47%**	37	3	0	272	0	**47%**	5	1	0	-	0	47%	5
GG1	72	499	190	38	0	499	0	5%	1	3	0	499	0	5%	15	3	0	499	0	5%	3	-	-	-	-	-	-
GG2	38	281	57	46	0	281	0	22%	7	3	0	60	0	22%	13	3	0	199	0	23%	8	1	0	281	0	22%	7
GG3	30	550	570	12	0	216	0	0%	3	3	0	16	0	58%	31	3	0	68	0	46%	13	2	0	216	0	0%	3
GG4	56	2,962	1,718	56	0	1,750	0	4%	1	3	0	796	0	43%	42	3	0	1,165	0	46%	11	-	-	-	-	-	-
GG5	30	192	1,900	19	0	55	0	74%	7	4	0	0	0	68%	23	3	0	36	0	**68%**	9	1	0	55	0	74%	7
TS1	30	269	1,477	16	0	269	0	0%	11	3	0	19	0	94%	38	3	0	161	0	53%	7	1	0	269	0	0%	11
TS2	40	367	1,881	13	0	365	0	0%	1	3	0	153	0	81%	42	3	0	256	0	31%	6	-	-	-	-	-	-
TS3	47	91	1,881	8	0	91	0	0%	1	5	0	0	0	88%	44	3	0	80	0	50%	10	-	-	-	-	-	-
TS4	71	520	1,313	31	0	520	0	0%	1	3	0	425	0	71%	44	3	0	259	0	0%	3	3	0	519	0	0%	44
TS5	68	235	1,000	14	0	137	0	64%	35	3	0	121	0	64%	49	3	0	160	0	35%	11	1	0	137	0	64%	35

the same random seed. To avoid biases in selecting the seeds, we used the numbers from 0 to 9. We executed each run on a dedicated Amazon® EC2 instance (*c5.4xlarge*) with 16 Intel® Xeon® 3.9GHz CPUs and 32GB of RAM. The total machine-time of the experiments was 1,530 hours (34 subjects x 3 tools x 90 minutes x 10 repetitions).

4.3 RQ1: Effectiveness

Columns "*initial assertion α*" of Table 4 indicate the quality of the initial assertions generated with Daikon. For eleven subjects (SE4, CM1, CM2, CM3, CM5, CL1, GG1, GG4, TS2, TS3 and TS4), Daikon does not generate any invariant, and in this case we consider *assert(true)* as the initial assertion. The false positives of α on the initial tests (FP_0) are always zero (except for subject CL5). This is an expected result because Daikon uses the execution traces of the initial tests to generate α. The size of the initial assertions ranges from 1 to 147 (28 on average). For six subjects the size is over 50, confirming that Daikon can generate many (often redundant) preconditions and postconditions [9]. The number of false negatives on the initial tests (FN_0) ranges from 0 to 1,750 (219 on average). The number of false positives on the validation tests (FP_v) is always zero, except for two subjects (SA5 and QA1), indicating that the initial assertions perform well with unseen tests. This result may depend on the high branch coverage of the EvoSuite-generated initial tests. The mutation score on the validation set ranges from 0% to 100% (38% on average).

Columns "*GAssert improved α' (median)*" indicate the quality of the GAssert improved assertions. We report the median values of the ten executions. The number of iterations ranges from 1 to 9 (3 on average). The median FP on the initial tests (FP_0) is zero for all subjects, as GAssert produces by construction assertions with zero false positives wrt $\mathbb{S}_0^+$. The median FN on the initial tests and mutations (FN_0) ranges from 0 to 1,282. The average is 125, which is 42.92% less than the average FN_0 of the initial assertion (= 219). For 24 subjects the median FN_0 of GAssert improved assertion α' is less than FN_0 of the initial assertion α. Although for nine subjects is the same (including subjects SE2, QA1 and QA4 with FN_0=0), GAssert often drastically reduced the size of the initial assertions. These results demonstrate that our evolutionary algorithm is effective in improving assertion oracles. For only subject SA2, α' has more false negatives than α. This is because when generating invariants Daikon relies on its own set of helper functions that are not supported by GAssert, and had thus to be excluded from the initial assertion when being passed to it for improvement.

The median FP on the validation set (FP_v) is zero for all subjects. The median mutation score on the validation set ($\mathcal{M}_v$%) ranges from 0% to 100% (58% on average)[1]. The $\mathcal{M}_v$% of α' is higher than the one of α for 16 subjects, with an increase of 34% on average.

[1]The values of $\mathcal{M}_v$% are in **underlined bold** if in at least one run the improved assertion has false positives (even after considering smaller assertions individually), and therefore we could not measure its mutation score. In such cases the value shown is the median of the runs with no false positives in the final assertion.

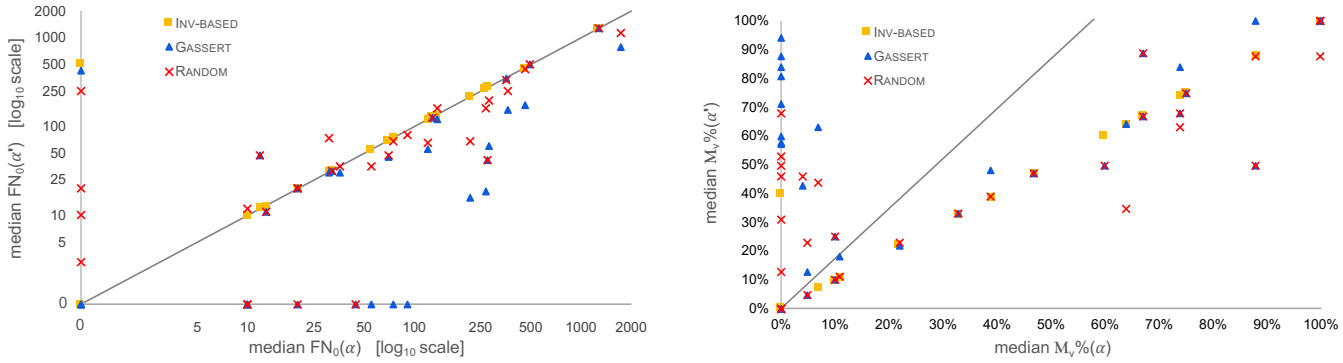

Figure 2: FN/Mutation score improvement comparison wrt the initial (left) and validation (right) sets.

4.4 RQ2: Comparison with Random and Invariant-Based Oracle Improvement

In this research question we compare GAssert with two baselines: GAssert with no guidance by the fitness functions (Random) and the invariant inference of Daikon (Inv-based). We set up the process so that these two baselines are used as part of the same iterative oracle improvement process of GAssert, described in Algorithm 1. The only difference among GAssert, Random and Inv-based is how each of them performs the oracle improvement process (line 6 of Algorithm 1). When running and evaluating Random and Daikon we used the same evaluation setup of RQ1.

Random is a variant of GAssert, in which there is no evolutionary pressure in the population because any guidance by the fitness functions is disabled. We obtained Random by modifying GAssert as follows: (i) we replaced the tournament and best-match selection with random selection; (ii) we disabled the Merging crossover and Mutation-based node selector; (iii) we disabled elitism and migration. Random terminates the random evolution of the two populations when either population finds a perfect assertion or the time budget expires. In the latter case, Random outputs the best assertion (wrt ϕ_{FP}) among all those generated so far.

Although *true random search* would have been the ideal baseline, enumerating and (uniformly) sampling the search space is infeasible because of the huge size of the search space. There are 1.978×10^{27} possible binary trees (assertion oracles) for each program point (Catalan number [44] with maximum tree size of 50). This is just a lower-bound of the search space because for each of these trees we need to consider all possible valid assignments of nodes to variables and functions. As such, we opted for a variant of GAssert that uses crossover and mutations operations to explore the search space, but without any guidance by the fitness functions.

Columns "*Random improved α′ (median)*" indicate the quality of the assertions returned by Random. The results show that the improved assertion of GAssert dominates the one of Random for 20 (59%) and 17 (50%) subjects considering the initial and validation sets, respectively. In such cases, GAssert assertions are substantially better than the one of Random. For 6 (18%) and 10 (25%) of subjects Random assertions outperform GAssert ones, but in this cases the difference is minimal. For the remaining cases the tools are showing similar results.

Inv-based is an oracle improvement approach that relies on dynamic invariant generation to improve oracle assertions. We chose Daikon (v.5.7.2) to build Inv-based because is the only publicly available tool that meets our requirements: (i) works with Java programs, (ii) generates executable Java-like assertions, (iii) takes in input a list of observer methods (for a fair comparison, it should use the same observer methods used by GAssert).

Daikon does not aim to improve a given assertion α nor relies on incorrect executions (FN). However, Daikon can rely on the test cases that OASIs outputs, which represent evidence of false positives of α. More specifically, the Inv-based oracle improvement process repeats the following two steps until the time budget expires, or it is not able to generate any invariant, or OASIs does not find any false positives for α: (i) execute the current test suite and compute the invariant α; (ii) invoke OASIs to get the test cases that reveal FPs for α and add them to the test suite.

Column "*Daikon improved α′ (median)*" indicate the quality of the assertions returned by Daikon. For ten subjects we did not run Inv-based because Daikon did not generate an initial assertion, and thus we compare GAssert and Inv-based with the remaining subjects. Considering the fitness function ϕ_{FP}, the improved assertion of GAssert dominates the one of Inv-based for 19 (59%) and 15 (63%) subjects considering the initial and evaluation sets, respectively. In such cases, GAssert assertions are substantially better than the one of Inv-based. For 2 (8%) and 7 (29%) subjects Inv-based assertions dominates GAssert ones considering the initial and evaluation sets, respectively. For the remaining cases nor GAssert or Inv-based assertions dominate each other.

Figure 2 plots the median FN_0 (left) and $\mathcal{M}_V\%$ (right) for each pair of initial and improved assertions wrt GAssert, Random and Inv-based. If a point is on the diagonal it means that the corresponding approach did not improve the false negatives or mutation score wrt the initial assertion. For the initial sets (left plot), most of GAssert points are under the diagonal, which means that GAssert produced improved assertion with less FNs. For the validation sets (right plot), most of GAssert points are above the diagonal, which means that GAssert produced improved assertion with higher mutation score. The plots also show that in most cases GAssert outperforms both Random and Inv-based oracle improvement.

Table 5: Evaluation Results for RQ3

subj.	Initial α			GAssert α'			Human α'						
ID	FP_υ	$\mathcal{M}_\mathbf{v}$	Size	FP_υ	$\mathcal{M}_\mathbf{v}$	Size	type	FP_υ	$\mathcal{M}_\mathbf{v}$	Size	Ov.	Exc.	GPB
SE1	523	-	3	0	75%	15	M	0	75%	7	73	25	12
SE2	0	75%	7	0	100%	17	M	0	100%	7	63	28	1
SE3	0	0%	1	0	33%	3	M	0	33%	7	52	0	5
SE4	0	40%	9	0	57%	30	M	0	57%	9	60	11	0
SA3	0	50%	7	0	50%	5	M	0	50%	7	14	0	3
SA4	0	0%	3	0	67%	7	M	0	67%	9	14	0	4
SA3	0	50%	7	0	50%	5	M + O	0	50%	11	15	0	0
SA4	0	0%	3	0	67%	7	M + O	1	67%	9	15	0	0

4.5 RQ3: Comparison with Human Oracle Improvement

Jahangirova et al. [22] conducted a human study to assess the ability of humans to improve assertion oracles. They performed this study in two settings: (i) the assertion is improved manually by humans without any tool support (M) (ii) the assertion is improved in an iterative setting with the use of OASIs (M + O). Overall, they recruited 29 humans to participate in the study. The subject methods they considered were SA3 and SA4 from the *StackAr* class. Moreover, the authors also share the data collected from Amazon® Mechanical Turk, which consists of manually improved assertions for four simple methods, performed by 74 different crowd-workers. As the results are publicly available [22], we compare these assertions with the ones produced by GAssert.

We run GAssert with the input assertions that were provided to the study participants. Then, as in RQ1 and RQ2, we measure the oracle deficiencies in the initial and GAssert improved assertions (wrt the validation set). We then compare these values to the oracle deficiencies in the assertions improved by humans. Column "*type*" of Table 5 indicates whether this improvement was purely manual (M) or included OASIs (M + O). As for four methods from our set the oracle improvement was performed by crowd-workers and no action was taken to ensure that they have a proper background or experience for such a task, we apply an additional filtering step to the list of assertions for these methods. We exclude the assertions that do not improve the initial assertion, i.e., they do not have less false positives or a higher mutation score. Column "*Ov.*" shows the overall number of assertions available and Column "*Exc.*" shows the number of assertions that were excluded.

The results show that GAssert is always able to improve the initial assertion and achieve a higher mutation score. Moreover, the median values across 10 runs for GAssert and across the number of human participants for manual improvement are always the same. Column "*GPB*" (GAssert Performs Better) reports the number of manual improvements that achieve a lower mutation score than GAssert does (10% of cases).

5 RELATED WORK

GAssert is the first fully-automated technique to improve oracle assertions. The closest related work is on invariant generation, oracle quality, and oracle improvement.

Invariant Generation. Dynamic invariant generators produce Boolean expressions (called program invariants) that evaluate to true for all the executions of an input test suite [6, 9, 10, 17, 32, 33, 37]. GAssert improves assertion oracles by reducing their false positives and false negatives, and as such can improve the assertions produced by invariant generators, which are known to be incomplete and imprecise when used as assertion oracles [6, 29, 40].

Ratcliff et al. proposed an evolutionary approach [35] for invariant generation that leverages negative counterexamples to rank the invariants. Differently from GAssert, their approach uses negative counterexamples in a post-processing phase and not as a part of the fitness function. Moreover, GAssert uses OASIs to actively generate positive and negative counterexamples. In addition, GAssert considers both externally (parameters, return values) and internally observable variables (local variables, private fields).

Oracle Quality Metrics. Research on measuring oracle quality mostly focuses on assertions in the test cases (test oracles) [19, 25, 38]. For instance, EvoSuite [11, 12] and a parameterized test case generator proposed by Fraser and Zeller [14] select from an initial set of possible assertions those that kill the highest number of mutations. These studies propose metrics to select test oracles with no guidance on how to improve them. GAssert focuses on assertions in the program, and not in the tests, evaluates the quality of oracles in terms of both false positives and false negatives, and actively improves program oracles by generating new assertions.

Oracle Improvement. Zhang et al.'s *iDiscovery* approach [46] improves the accuracy and completeness of invariants by iterating a feedback loop between Daikon and symbolic execution. The invariants generated by *iDiscovery* are still limited within the set of Daikon templates. Therefore they are not as expressive as the ones generated with GAssert. OASIs [20, 21] relies on humans to improve a given oracle assertion so that it does not suffer from the reported oracle deficiencies. Given oracle deficiencies identified by OASIs, GAssert automates the difficult task of improving assertions with a novel evolutionary algorithm.

6 CONCLUSION

Improving assertion oracles is important to increase the fault-detection capabilities of both manually written and automatically generated test cases [4]. In this paper, we presented GAssert, the first automated approach to improve assertion oracles.

Our experiments indicate that GAssert improved assertions has zero false positives, and the number of false negatives in them is largely reduced with respect to the initial Daikon assertions. The few sample cases with independently obtained human improvements indicate that GAssert is competitive with – and even sometime better than – human improvements.

We plan in the future to increase the expressiveness of GAssert assertions by also considering the universal and existential quantifiers. Also, we plan to investigate how difficult is for a developer to understand the assertions produced by GAssert.

ACKNOWLEDGEMENTS

This work was partially supported by the Swiss SNF project ASTERIx (SNF 200021_178742) and the H2020 project PRECRIME, funded under the ERC Advanced Grant 2017 Program (ERC Grant Agreement n. 787703).

REFERENCES

[1] Shay Artzi, Michael D. Ernst, Adam Kieżun, Carlos Pacheco, and Jeff H. Perkins. 2006. Finding the Needles in the Haystack: Generating Legal Test Inputs for Object-Oriented Programs. In *Workshop on Model-Based Testing and Object-Oriented Systems (M-TOOS '06)*.

[2] Angello Astorga, P. Madhusudan, Shambwaditya Saha, Shiyu Wang, and Tao Xie. 2019. Learning Stateful Preconditions Modulo a Test Generator. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI '19)*. ACM, 775–787.

[3] Thomas Back. 1996. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford University Press.

[4] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525.

[5] Markus F Brameier and Wolfgang Banzhaf. 2007. A Comparison with Tree-Based Genetic Programming. *Linear Genetic Programming* (2007), 173–192.

[6] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. 2008. DySy: Dynamic Symbolic Execution for Invariant Inference. In *Proceedings of the International Conference on Software Engineering (ICSE '08)*. ACM, 281–290.

[7] Jason M. Daida, Adam M. Hilss, David J. Ward, and Stephen L. Long. 2003. Visualizing Tree Structures in Genetic Programming. In *Proceedings of the conference on Genetic and Evolutionary Computation (GECCO '03)*. Springer, 1652–1664.

[8] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. 2002. A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation* 6, 2 (2002), 182–197.

[9] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. 1999. Dynamically Discovering Likely Program Invariants to Support Program Evolution. In *Proceedings of the International Conference on Software Engineering (ICSE '99)*. ACM, 213–224.

[10] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. 2001. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE Transactions on Software Engineering* 27, 2 (2001), 99–123.

[11] Gordon Fraser and Andrea Arcuri. 2011. Evolutionary Generation of Whole Test Suites. In *Proceedings of the International Conference on Quality Software (QSIC '11)*. IEEE, 31–40.

[12] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE '11)*. ACM, 416–419.

[13] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Transactions on Software Engineering* 39, 2 (2013), 276–291.

[14] Gordon Fraser and Andreas Zeller. 2011. Generating Parameterized Unit Tests. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '11)*. ACM, 364–374.

[15] Juan Pablo Galeotti, Carlo A. Furia, Eva May, Gordon Fraser, and Andreas Zeller. 2014. DynaMate: Dynamically Inferring Loop Invariants for Automatic Full Functional Verification. In *Proceedings of the Haifa Verification Conference (HVC '14)*. Springer, 48–53.

[16] Juan P. Galeotti, Carlo A. Furia, Eva May, Gordon Fraser, and Andreas Zeller. 2015. Inferring Loop Invariants by Mutation, Dynamic Analysis, and Static Checking. *IEEE Transactions on Software Engineering* 41, 10 (2015), 1019–1037.

[17] Ashutosh Gupta and Andrey Rybalchenko. 2009. Invgen: An Efficient Invariant Generator. In *Proceedings of the International Conference on Computer Aided Verification (CAV '09)*. Springer, 634–640.

[18] Mark Harman, William B. Langdon, Yue Jia, David Robert White, Andrea Arcuri, and John A. Clark. 2012. The GISMOE Challenge: Constructing the Pareto Program Surface using Genetic Programming to Find Better Programs (keynote paper). In *Proceedings of the International Conference on Automated Software Engineering (ASE '14)*. ACM, 1–14.

[19] Chen Huo and James Clause. 2014. Improving Oracle Quality by Detecting Brittle Assertions and Unused Inputs in Tests. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '14)*. ACM, 621–631.

[20] Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella. 2016. Test Oracle Assessment and Improvement. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '16)*. ACM, 247–258.

[21] Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella. 2018. OASIs: Oracle Assessment and Improvement Tool. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '18)*. ACM, 368–371.

[22] Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella. 2019. An Empirical Validation of Oracle Improvement. *IEEE Transactions on Software Engineering* (2019).

[23] René Just. 2014. The Major Mutation Framework: Efficient and Scalable Mutation Analysis for Java. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '14)*. ACM, 433–436.

[24] John R Koza and John R Koza. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Vol. 1. MIT press.

[25] William B. Langdon, Shin Yoo, and Mark Harman. 2017. Inferring Automatic Test Oracles. In *Proceedings of the International Workshop on Search-Based Software Testing (SBST '17)*. IEEE, 5–6.

[26] Y. Lavinas, C. Aranha, T. Sakurai, and M. Ladeira. 2018. Experimental Analysis of the Tournament Size on Genetic Algorithms. In *International Conference on Systems, Man, and Cybernetics (SMC '18')*. IEEE, 3647–3653.

[27] David Lo and Shahar Maoz. 2009. Mining Scenario-Based Specifications with Value-Based Invariants. In *Proceedings of the International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, 755–756.

[28] Brad L. Miller, Brad L. Miller, David E. Goldberg, and David E. Goldberg. 1995. Genetic Algorithms, Tournament Selection, and the Effects of Noise. *Complex Systems* 9, 3 (1995), 193–212.

[29] Cu D. Nguyen, Alessandro Marchetto, and Paolo Tonella. 2013. Automated Oracles: An Empirical Study on Cost and Effectiveness. In *Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE '13)*. ACM, 136–146.

[30] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *Proceedings of the International Conference on Software Engineering (ICSE '07)*. ACM, 75–84.

[31] Annibale Panichella, Rocco Oliveto, Massimiliano Di Penta, and Andrea De Lucia. 2015. Improving Multi-Objective Test Case Selection by Injecting Diversity in Genetic Algorithms. *IEEE Transactions on Software Engineering* 41, 4 (2015), 358–383.

[32] Corina S. Pasareanu and Willem Visser. 2004. Verification of Java Programs Using Symbolic Execution and Invariant Generation. In *Proceedings of the International SPIN Workshop on SPIN Model Checking and Software Verification (SPIN '04)*. Springer, 164–181.

[33] Long H. Pham, Jun Sun, Lyly Tran Thi, Jingyi Wang, and Xin Peng. 2017. Learning Likely Invariants to Explain Why a Program Fails. In *Proocedings of the International Conference on Engineering of Complex Computer Systems (ICECCS '17)*. IEEE, 70–79.

[34] Dipesh Pradhan, Shuai Wang, Shaukat Ali, Tao Yue, and Marius Liaaen. 2017. CBGA-ES: A Cluster-Based Genetic Algorithm with Elitist Selection for Supporting Multi-Objective Test Optimization. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST '17)*. IEEE, 367–378.

[35] Sam Ratcliff, David R. White, and John A. Clark. 2011. Searching for Invariants Using Genetic Programming and Mutation Testing. In *Proceedings of the conference on Genetic and Evolutionary Computation (GECCO '11)*. ACM, 1907–1914.

[36] Henry G. Rice. 1953. Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society* 74, 2 (1953), 358–366.

[37] Abhik Roychoudhury and I. V. Ramakrishnan. 2004. Inductively Verifying Invariant Properties of Parameterized Systems. *Automated Software Engineering* 11, 2 (2004), 101–139.

[38] D. Schuler and A. Zeller. 2011. Assessing Oracle Quality with Checked Coverage. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST '11)*. 90–99.

[39] Oren Shoval, Hila Sheftel, Guy Shinar, Yuval Hart, Omer Ramote, Avi Mayo, Erez Dekel, Kathryn Kavanagh, and Uri Alon. 2012. Evolutionary Trade-Offs, Pareto Optimality, and the Geometry of Phenotype Space. *Science* 336, 6085 (2012), 1157–1160.

[40] Matt Staats, Shin Hong, Moonzoo Kim, and Gregg Rothermel. 2012. Understanding User Understanding: Determining Correctness of Generated Program Invariants. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '12)*. ACM, 188–198.

[41] Hisashi Tamaki, Hajime Kita, and Shigenobu Kobayashi. 1996. Multi-Objective Optimization by Genetic Algorithms: A Review. In *Proceedings of IEEE International Conference on Evolutionary Computation*. IEEE, 517–522.

[42] Tao Xie, D. Notkin, and D. Marinov. 2004. Rostra: a Framework for Detecting Redundant Object-Oriented Unit Tests. In *Proceedings of the International Conference on Automated Software Engineering (ASE 04)*. 196–205.

[43] Shuai Wang, Shaukat Ali, Tao Yue, and Marius Liaaen. 2018. Integrating Weight Assignment Strategies With NSGA-II for Supporting User Preference Multiobjective Optimization. *IEEE Transation on Evolutionary Computation* 22, 3 (2018), 378–393.

[44] Julian West. 1995. Generating Trees and the Catalan and Schröder Numbers. *Discrete Mathematics* 146, 1-3 (1995), 247–262.

[45] Darrell Whitley. 1994. A Genetic Algorithm Tutorial. *Statistics and Computing* 4, 2 (1994), 65–85.

[46] Lingming Zhang, Guowei Yang, Neha Rungta, Suzette Person, and Sarfraz Khurshid. 2014. Feedback-driven dynamic invariant discovery. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '14)*. ACM, 362–372.