European Research Council
Established by the European Commission

Self-assessment Oracles for Anticipatory Testing

TECHNICAL REPORT: TR-Precrime-2020-02

Valerio Terragni, Gunel Jahangirova, Mauro Pezzè, Paolo Tonella
Università della Svizzera italiana

Evolutionary Improvement of Assertion Oracles

Project no.:	787703
Funding scheme:	ERC-2017-ADG
Start date of the project:	January 1, 2019
Duration:	60 months

Technical report num.:	TR-Precrime-2020-02
Date:	January 2020
Organization:	Università della Svizzera italiana
Authors:	Valerio Terragni, Gunel Jahangirova, Mauro Pezzè, Paolo Tonella Università della Svizzera italiana
Dissemination level:	Public
Revision:	1.0

Disclaimer:
This Technical Report is a pre-print of the following publication:

Valerio Terragni, Gunel Jahangirova, Mauro Pezzè, Paolo Tonella: *Evolutionary Improvement of Assertion Oracles*. Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), Sacramento, USA, November, 2020

Please, refer to the published version when citing this work.

Università della Svizzera Italiana (USI)

Principal investigator:	Prof. Paolo Tonella
E-mail:	paolo.tonella@usi.ch
Address:	Via Buffi, 13 – 6900 Lugano – Switzerland
Tel:	+41 58 666 4848
Project website:	https://www.pre-crime.eu/

Abstract

Assertion oracles are executable boolean expression placed inside the program. A perfect assertion oracle should pass (return true) for all correct executions and fail (return false) for all incorrect executions. The difficulty of manually design or automatically generate perfect oracles often leads to assertions that fail to distinguish between correct and incorrect executions. In other words, they have false positives and false negatives that we call oracle deficiencies.

In this paper, we propose GASSERT (Genetic ASSERTion improvement), the first technique to automatically improve assertions oracles. Given an assertion oracle and its oracle deficiencies, GASSERT uses a novel evolutionary algorithm that explores the space of possible assertions to identify a new assertion with fewer oracle deficiencies than the original assertion.

Our empirical evaluation on 34 JAVA methods from 7 JAVA code bases shows that GASSERT effectively improves assertion oracles. Moreover, GASSERT outperforms two baselines (unguided-random and invariant-based oracle improvement) and was competitive with and in some cases even outperformed human improved assertions.

Contents

1 Introduction

While recent progresses in test case generation produced effective techniques and tools, capable of achieving high code coverage [15, 33, 53, 50], the oracle problem still represents a major obstacle to the widespread adoption of automated testing approaches [20, 5, 30].

A commonly used oracle in software testing consists of program assertions, i.e., executable boolean expressions placed inside the program that predicates on the values of variables at a specific program point. A perfect oracle should pass (return true) for all correct executions and fail (return false) for all incorrect executions. Perfect oracles are often difficult to design, and assertion oracles often fail to distinguish between correct and incorrect executions [30], that is, they are prone to both false positives and false negatives [23]. 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). False positives and false negatives are jointly called *oracle deficiencies*.

Oracle deficiencies are a serious problem also for invariant generators, which are known to produce invariants that are incomplete and imprecise when used as assertion oracles [6, 36, 47]. They are incomplete because most dynamic invariant generators, notably DAIKON [10, 11] and INVGEN [19], rely on a set of pre-defined templates of Boolean expressions to produce program invariants, and cannot generate assertions that do not match the templates [6]. As such they can explore only a limited portion of the search space of all possible assertions. Moreover, most invariant generators (such as DAIKON) ignore internally observable variables (method-local variables and private fields). This further reduces the expressiveness of the assertions they generate. Existing invariant generators are also imprecise, because the invariants they generate automatically often do not generalize well with unseen test cases. In fact, Nguyen et al.'s and Staats et al.'s studies [36, 47] report high false positive rates for DAIKON invariants. One of the reasons is that invariant generators produce invariants using only positive counterexamples [10, 11, 19]. Moreover, by ignoring incorrect execution states (false negatives) there is no guarantee that the resulting assertions will be effective in exposing software faults.

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 the OASIS approach [24, 25] to automatically identify evidence of oracle deficiencies in program assertions. Given a program assertion, OASIS generates test cases and mutations that indicate the presence of false positives and false negatives. The found evidence is meant to support the developers in assessing and improving assertion oracles.

A recent study by OASIS's authors shows that manual improvement of assertion oracles is difficult [26]. Given the oracle deficiencies detected by OASIS, for only 67% of the given assertions humans successfully removed all oracle deficiencies. Staats et al. [47] confirmed that for humans is hard reasoning about soundness and completeness of program assertions. Their study shows that developers often fail to find false positives in DAIKON invariants.

The difficulty of manually improving assertion oracles motivated us to study the problem of an automated improvement: given a program assertion α and some evidence of the presence of false positives and false negatives provided by an oracle assessor $\mathcal{OA}$ (such as OASIS), we want to automatically generate an improved assertion α' with fewer oracle deficiencies than α. While there are many techniques to automatically generate program assertions, e.g., program invariants [18, 17, 43, 54, 32, 9, 19, 41, 1], automatically improving assertions oracles is a largely unexplored problem.

In this paper, we propose **GASSERT**, ***Genetic ASSERTion improvement***, the first technique to automatically improve assertions 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 manually removing false alarms in assertions is costly and difficult [36].

GASSERT addresses the challenge of a huge search space with an evolutionary approach that evolves populations of assertions by rewarding assertions with fewer deficiencies. GASSERT initializes the populations of assertions by mutating the original assertion, so as to produce "good" genetic material for evolution. GASSERT formulates the oracle improvement problem as a multi-objective optimization problem (MOOP) [48] 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, i.e., the number of variables and function calls in it.

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 [48, 40, 37, 44]. 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 negative, 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.

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% with respect to the mutation score obtained with the initial assertions. This paper makes the following contributions:

- We formulate the problem of automatically improving assertion oracles given some evidence of the presence of false positives and false negatives,
- We propose GASSERT, an evolutionary approach to automatically improve a given assertion oracle by reducing its false positives and false negatives,
- 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 and tool to facilitate future work in this area `http://tiny.cc/jkqxkz`

2 Problem Formulation

This section provides the preliminaries for this work and formulates the problem of improving oracle assertions.

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 $\rho\rho$ and 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. An assertion oracle α aims to distinguish between correct and incorrect executions. More specifically, an *assertion oracle* 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 an assertion that meets such a condition a *perfect* oracle.

We consider oracle assertions inserted into program $\mathcal{P}$, and not into its test cases. The difference is that assertions in $\mathcal{P}$ handle all the test case executions, while assertions in the test cases check the correctness of executions of a single (possibly parameterized) test.Perfect oracles are difficult to design, and assertion oracles often fail to distinguishing between correct and incorrect executions. In other words, they are prone to false positives and false negatives, which we call *oracle 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

Definition 1 *A **false positive** of an assertion α at a program point pp 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 $s^+ \in \mathcal{S}^+_{pp}$: $\alpha[s^+] = F$.*

Definition 2 *A **false negative** of an assertion α at a program point pp 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 $s^- \in \mathcal{S}^-_{pp} : \alpha[s^-] = T$.*

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

Identifying oracle deficiencies by enumerating all correct and incorrect states is infeasible, because it requires to enumerate infinitely many executions [42]. 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. An oracle assessor $\mathcal{OA}$ can be a human or an automated technique. We assume $\mathcal{OA}$ to be *precise*, that is, it reports only real oracle deficiencies, but possibly *incomplete*, that is, it may miss oracle deficiencies, because it cannot enumerate all possible correct and incorrect executions. To enable full automation, in this paper we use OASIs [23, 26] as an $\mathcal{OA}$. Given an assertion α, OASIs leverages search-based test generation and mutation testing to report evidence of false positives and false negatives, if any can be found in the given time budget.

OASIs finds false positives of an assertion α by generating test cases that reach the program point of α, and make α return false in the reached state. Such states are false positives if the behavior of the program is correct (otherwise, OASIs discovered a fault in the program, not in the assertion) [23, 26].

OASIs finds false negatives for an assertion α by seeding artificial faults (mutations) into program $\mathcal{P}$ with mutation testing [15]. OASIs generates a test case and a mutation that produce a corrupted program state $s^- \in \mathcal{S}^-$ at the assertion point pp, where α does not reveal the fault, i.e., its outcome is 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 (the number of variables and functions occurring in the assertions) to a maximum value (50 in our experiments).

Let FP$(\alpha, \mathbb{S}^+)$ denote the number of false positives of α wrt a finite subset $\mathbb{S}^+$ of $\mathcal{S}^+$, i.e., FP$(\alpha, \mathbb{S}^+)$ is the number of states $s^+ \in \mathbb{S}^+ \subseteq \mathcal{S}^+ : \alpha[s^+] = F$. Similarly, FN$(\alpha, \mathbb{S}^-)$ denotes the number of false negatives of α wrt a finite subset $\mathbb{S}^-$ of $\mathcal{S}^-$, i.e., 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 pp 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 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$(\alpha, \mathbb{S}^-)$ within $\mathcal{B}$.*

In defining oracle improvement, we give priority to false positive over false negative reduction, by requiring all false positives to disappear in the improved oracle. The rationale for this choice is that assertions that easily reduce false negatives likely introduce many false alarms. Such assertions trigger an expensive debugging process since the root of those bugs may likely be the assertion itself. Therefore we privilege assertions with no false alarms.

The ideal improved assertion oracle α' has zero oracle deficiencies wrt to $\mathbb{S}^+$ and $\mathbb{S}^-$ (FP$(\alpha', \mathbb{S}^+) =$ FN$(\alpha', \mathbb{S}^-) =$ 0). However, generating ideal assertions is 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 [23]. Therefore, an oracle with zero false positives and the lowest number of false negatives is deemed sufficiently adequate in practice [26].

Algorithm 1: GASSERT, iterative oracle improvement process

```
input  : initial assertion $\alpha$ at progr. point $\rho\rho$ in $\mathcal{P}$, time-budget $\mathcal{B}$
output: improved assertion $\alpha'$
1  function GASSERT
2      $\mathcal{P}' \leftarrow$ INSTRUMENT-METHOD-AT-PROGRAM-POINT($\rho\rho$, $\mathcal{P}$)
3      $\langle \mathbb{S}^+, \mathbb{S}^- \rangle \leftarrow$ GET-INITIAL-CORRECT-AND-INCORRECT-STATES($\mathcal{P}'$)
4      while time-budget $\mathcal{B}$ is not expired do
5          $\Sigma \leftarrow$ GET-DICTIONARY-OF-VARIABLES($\mathbb{S}^+, \mathbb{S}^-$)
6          $\alpha' \leftarrow$ ORACLE-IMPROVEMENT($\alpha, \mathbb{S}^+, \mathbb{S}^-, \Sigma$)
7          $\langle \mathbb{S}^+_{\text{NEW}}, \mathbb{S}^-_{\text{NEW}} \rangle \leftarrow$ ORACLE-ASSESSMENT($\alpha'$)          // OASIs [24]
8          if $\mathbb{S}^+_{\text{NEW}} = \varnothing \wedge \mathbb{S}^-_{\text{NEW}} = \varnothing$ then
9              return $\alpha'$
10         $\mathbb{S}^+ \leftarrow \mathbb{S}^+ \cup \mathbb{S}^+_{\text{NEW}}$
11         $\mathbb{S}^- \leftarrow \mathbb{S}^- \cup \mathbb{S}^-_{\text{NEW}}$
12         $\alpha \leftarrow \alpha'$
13     return $\alpha'$
```

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 $(min \leq y)$	$\varnothing$	$\varnothing$	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 $(min \leq y))$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$

3 GASSERT

In this section, we overview the GASSERT approach and illustrate the GASSERT oracle improvement process with a running example.

Overview. GASSERT improves oracle assertions with an iterative process (see Algorithm 1). GASSERT's required inputs are an oracle assertion α, the program point $\rho\rho$ in $\mathcal{P}$ where α is placed, and a time budget $\mathcal{B}$. GASSERT's output is an improved assertion α'.

GASSERT starts by instrumenting $\mathcal{P}$ to capture program states at runtime (line 2 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 its faulty versions (i.e., mutants), respectively (line 3 Algorithm 1). The while loop (lines 4–13) implements the iterative process, within which GASSERT creates the dictionary of variables Σ from the states $\mathbb{S}^+$ and $\mathbb{S}^-$ and uses these variables to build new oracle assertions (line 5 Algorithm 1). The ORACLE-IMPROVEMENT algorithm that we discuss in detail in Section 4 returns an improved assertion α' (line 6 of Algorithm 1). If $\mathcal{OA}$ cannot find any oracle deficiencies of α', Algorithm 1 returns α', and the process terminates (lines 8 and 9 Algorithm 1). 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 of Algorithm 1). The improved assertion α' replaces the initial assertion α (line 12 Algorithm 1) and a new iteration starts.

3.1 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 shows also (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 identify FN (lines 6 and 4).

Table 2 illustrates how GASSERT improves an incomplete and trivially wrong 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 $(min < x)$ 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 produced 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 produced them. On such states α passes but it should fail.

```
public static int min(int x, int y) {
 serializer(x, y); // instrumentation
 int min;
 if (x <= y) {
     min = x; // mutant M2 : min = 0;
 } else {
     min = y; // mutant M1 : min = y + 1;
 }
 serializer(x, y, min); // instrumentation
 // program point pp
 return min;
}
```

Figure 1: Java source code of the running example

In the example, $\mathcal{OA}$ identifies both false positives and false negatives for α : *min* $< x$. Table 2 reports a sample test case t_1 that $\mathcal{OA}$ generates for α and for which α incorrectly returns false. The test cases t_1 produces the state s_1^+ that is a false positive for α (see Def. 1). The table reports also 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 the *killable* mutant M_1. The initial assertion α wrongly returns true for the faulty program state s_2^-, thus s_2^- 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 α' : *min* $\leq x$ AND *min* $\leq y$. GASSERT produces the new assertion with an evolutionary algorithm that evolves populations of assertions towards an assertion with zero false positives and the lowest number of false negatives. More specifically, 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, (iii) mutating the offspring (with a certain probability) using mutation operators.

Let us consider the initial assertion α : *min* $< x$, and let us assume that GASSERT selects two parents: α_{p1} : *min* $\leq x$ and α_{p2} : *min* $\neq y$. The assertion α_{p1} reduces the number of false positives with respect to the initial assertion (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 s_2^- (FN$(\alpha_{p1}, \mathbb{S}^-) = 1$, where $\mathbb{S}^- = \{s_2^-\}$). On the contrary, the assertion α_{p2} reduces the number of false negatives (FN$(\alpha_{p2}, \mathbb{S}^-) = 0$), but it has the same number of false positives as α (FP$(\alpha_{p2}, \mathbb{S}^+) =$ FP$(\alpha, \mathbb{S}^+) = 1$).

The crossover operator *merge crossover* applied to α_{p1} and α_{p2} would produce the offspring α_{o1} : (*min* $\leq x$) AND (*min* $\neq y$) and α_{o2} : (*min* $< x$) OR (*min* $\neq y$). If the mutation operator *node mutation* mutates α_{o1} into (*min* $\leq x$) AND (*min* $\leq y$), GASSERT would obtain an improved assertion with zero oracle deficiencies wrt the states $\mathbb{S}^+$ and $\mathbb{S}^-$, and the first iteration would terminate.

At the second iteration, $\mathcal{OA}$ takes in input α : *min* $\leq x$ AND *min* $\leq y$ to find oracle deficiencies. For this assertion, $\mathcal{OA}$ does not find false positives, but it reports a false negative: execution of test t_3 with mutant M_2 leads to the state s_3^-, which is a false negative for α (i.e., FN$(\alpha, \mathbb{S}^-) = 1$, where $\mathbb{S}^- = \{s_2^-, s_3^-\}$).

Given the assertion α : (*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 successfully finds (with crossover and mutation) the improved assertion α' : ((*min* $== x$) OR (*min* $== y$)) AND ((*min* $\leq x$) AND (*min* $\leq y$)), with zero oracle deficiencies (FP$(\alpha', \mathbb{S}^+) =$ FN$(\alpha', \mathbb{S}^-) = 0$). As $\mathcal{OA}$ does not find any oracle deficiencies for α', the improvement process terminates.

The following two sections describe in detail how GASSERT serializes program states and how improves the oracle assertions.

3.2 Program State Serialization

A program state is a set of variables $s = \{v_1, \cdots, v_n\}$ 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). Selecting the variables that a program state should contain is a key design decision as it defines the expressiveness of the assertions that GASSERT can produce. Given a method $m(\vec{p})$ with formal parameters $\vec{p}$, GASSERT constructs the program state at *pp* 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 names) and immediately before $\rho\rho$. By considering "old" values, GASSERT can generate assertions that also predicate on methods preconditions [9].

When the considered variable is a primitive, GASSERT simply adds it to the program state (rounding floats with a fixed precision). When variable v_i is not a primitive but an object, there are two well-established approaches to obtain primitive values: *Object serialization* [49] captures the values of all primitive-type object fields that are recursively reachable from v_i. *Observer methods* [1] are side-effect free methods on v_i that return a primitive value.

Object serialization can introduce many variables into state s, thus increasing the search space, which makes finding oracle improvements more difficult. Indeed, many recursively obtained primitive variables often do not capture interesting properties of methods. The effectiveness of the second approach relies on the availability of user-defined observer methods that capture all important aspects of the object under test.

Hybrid State Serialization. To address the aforementioned problem, GASSERT opts for a hybrid solution that uses both approaches. We rely on observer methods for all non-primitive variables considered by GASSERT. We use object serialization only for the object receiver (`this`) of the method under analysis m. For non-primitive fields of the object receiver we again consider observer methods. 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.

If *id*(v_i) = `this`, GASSERT serializes the object by adding variable `this.`*field*$_j$ to the state s, for each primitive-type field *field*$_j$ of `this`. For each non-primitive variable v_i (including `this`), GASSERT gets the observer methods $\{f_1, f_2, \ldots, f_m\}$ of v_i's class C using a static analyzer that scans the bytecode instructions of the public methods in C. GASSERT marks a method m as an *observer*, if (i) m returns a number or a boolean, and (ii) m cannot directly or indirectly execute `putfield` or `putstatic` bytecode instructions, i.e., m is side-effect free; (iii) m does not have additional parameters (besides the object receiver). For each observer method f_j with return type τ_j, the analyzer updates the state s by adding 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 (i.e., 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.

State Collection. Function INSTRUMENT-METHOD-AT-PROGRAM-POINT instruments $\mathcal{P}$ at the beginning and one immediately before $\rho\rho$ (Algorithm 1, line 2). When a test execution reaches $\rho\rho$ in the instrumented program, GASSERT performs the state serialization described above. Every time GASSERT executes a new test, it stores the observed states so that it 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 of 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. that the seeded fault (the mutant) has successfully corrupted the program state. If not, GASSERT found an *equivalent mutant* [27] which it then removes from $\mathbb{S}^-$.

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. GASSERT 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.

4 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) [2, 52]. We formulate the oracle improvement problem as a multi-objective optimization problem (MOOP) [48, 40, 37, 44] 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, i.e., 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, 51], rely on *Pareto optimality* [46, 48, 21] to produce solutions that offer the best trade-off between competing objectives [48]. 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* [52] 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}), respectively. These populations periodically exchange their best individuals (population migration) to introduce promising genetic material for the evolution in both $Popul^{FP}$ and $Popul^{FN}$. Eventually $Popul^{FP}$ will more likely produce assertions with zero false positives and fewer false negatives. In fact, the periodic migration of best individuals guarantees that $Popul^{FP}$ works on assertions with a decreasing amount of false negatives.

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 determined by the concept of *dominance* ($\prec$) [8]. While the standard definition of dominance gives the same importance to all objectives being optimized, we need an unbalanced definition of dominance towards false positives and negatives, respectively, that we define as follows:

Definition 3 ***FP-fitness* (ϕ_{FP}).** *Given two solutions α_1 and α_2 and two sets of correct $\mathbb{S}^+$ and incorrect $\mathbb{S}^-$ states, α_1 **dominates$_{\mathbf{FP}}$** α_2 ($\prec_{FP}$) 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 solutions α_1 and α_2 and two sets of correct $\mathbb{S}^+$ and incorrect $\mathbb{S}^-$ states, α_1 **dominates$_{\mathbf{FN}}$** ($\prec_{FN}$) α_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 ($FP(\alpha_1, \mathbb{S}^+) = FP(\alpha_2, \mathbb{S}^+) \wedge FN(\alpha_1, \mathbb{S}^-) = FN(\alpha_2, \mathbb{S}^-)$), both functions give preference to small assertions in terms of amount of variables and terms. 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

Algorithm 2: Evolutionary algorithm for the oracle improvement

```
input  : correct $\mathbb{S}^+$ and incorrect $\mathbb{S}^-$ states, assertion oracle $\alpha$
output: improved assertion oracle$\alpha'$
1  function ORACLE-IMPROVEMENT
2      $Popul^{FP}$ ← GET-INITIAL-POPULATION($\alpha$, $\Sigma$)
3      $Popul^{FN}$ ← GET-INITIAL-POPULATION($\alpha$, $\Sigma$)
4      for gen from 1 to max-number-of-gen do
5          if $\exists a' \in \{Popul^{FP} \cup Popul^{FN}\}$:$FP(a', \mathbb{S}^+) = FN(a', \mathbb{S}^-) = 0$
6          AND $gen \leq$ min-number-of-gen then
7              return $a'$                                        // optimal assertion
8          do in parallel
9              $Popul^{FP}$←SELECT-AND-REPRODUCE($Popul^{FP}$, $\phi_{FP}$,$\Sigma$,$\mathbb{S}^+$,$\mathbb{S}^-$)
10             $Popul^{FN}$←SELECT-AND-REPRODUCE($Popul^{FN}$,$\phi_{FN}$,$\Sigma$,$\mathbb{S}^+$,$\mathbb{S}^-$)
11         if gen % frequency-migration = 0 then
12             do in parallel
13                 $Popul^{FP}$ ← MIGRATE($Popul^{FN}$, $\phi_{FP}$, $\phi_{FN}$ )
14                 $Popul^{FN}$ ← MIGRATE($Popul^{FP}$, $\phi_{FN}$, $\phi_{FP}$)
15     return $a'$ with zero FP and the lowest number of FN
input  : population Popul, fitness function $\phi$, dictionary of variables $\Sigma$, correct $\mathbb{S}^+$ and incorrect $\mathbb{S}^-$ states,
         generation gen
output: new population $Popul_{\text{NEW}}$
16 function SELECT-AND-REPRODUCE
17     Popul ← COMPUTE-FITNESS(Popul, $\mathbb{S}^+$, $\mathbb{S}^-$)
18     $Popul_{\text{NEW}}$ ← $\varnothing$
19     if gen % frequency-of-elitism = 0 then
20         $Popul_{\text{NEW}}$ ← GET-BEST-INDIVIDUALS($\phi$)
21     while $Popul_{\text{NEW}}$ is not full do
22         $\langle a_{p1}, a_{p2} \rangle$ ←SELECT-PARENTS(Popul, $\phi$)
23         $\langle a_{o1}, a_{o2} \rangle$ ←CROSSOVER-AND-MUTATION($a_{p1}, a_{p2}$,$\Sigma$)
24         add $\langle a_{o1}, a_{o2} \rangle$ to $Popul_{\text{NEW}}$
25     return $Popul_{\text{NEW}}$
```

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 and τ.

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 [4]. 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 [4]. 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 a randomly-generated subtree (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 minimum 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 such cases, GASSERT returns the best archived assertion, the one with zero false positives and the lowest number of false negatives, i.e., 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) implementing a classic selection–crossover–mutation evolutionary approach [52]. Our problem requires the definition of novel selection and crossover operators, that are specific for automatic oracle improvement.

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 computation of *FP*$(\alpha, \mathbb{S}^+)$ and *FN*$(\alpha, \mathbb{S}^-)$ by (i) loading the states in the memory, (ii) computing multiple fitness values in parallel threads, (iii) caching the results for each assertion α, to avoid recomputing them upon encountering α multiple times during the evolution.

Function SELECT-AND-REPRODUCE initializes the new population Popul$_{\text{NEW}}$ with the empty set (line 18 of Algorithm 2) performing elitism if gen % frequency-migration = 0. Then it 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 [34] is a classic GP selection criterion [52]. It runs "tournaments" among K randomly-chosen individuals and selects the winner of each tournament (the one with the highest fitness) [34]. 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 [31]) as it mitigates the *local optima problem* [52].

Best-match Selection is a new criterion that we define in this paper, specific to our problem. The best-match selection criterion exploits semantic information about the identity of the correct and incorrect states that each individual *covers*. Let *cov*$^+(\alpha,\mathbb{S}^+)$ denote the subset of $\mathbb{S}^+$ on which α evaluates to true, i.e., *cov*$^+(\alpha,\mathbb{S}^+)$ $= \{s^+ \in \mathbb{S}^+ : \alpha[s^+] = T\} \subseteq \mathbb{S}^+$, and *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. Then, it 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. To guarantee genetic diversity, the criterion does not to always pick the assertion with highest weight. Symmetrically, if *Popul* is *Popul*^{FN}, the best-match criterion considers *cov*$^+$ instead of *cov*$^-$. Intuitively, the criterion increases the chances of crossover two complementary individuals that are likely to yield a fitter offspring.

Crossover. Function CROSSOVER-AND-MUTATION exchanges genetic material between the two parents α_{p1} and α_{p2}, producing two offspring α_{o1} and α_{o2} (line 23 in Algorithm 2),which are mutated (with a given probability) by means of the mutation operators used to initialize the two populations. GASSERT implements two different crossover operators, subtree crossover and merging crossover, and choses between them with a given probability.

Subtree Crossover [29] is the canonical tree-based crossover. Given two parents, the subtree crossover operator 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 [29].

Merging Crossover. Given two parents α_{p1} and α_{p2}, the merging crossover operator selects two boolean subtrees, α_1 from α_{p1} and α_2 from α_{p2}, and produces two offsprings α_{o1} and α_{o2} such that $\alpha_{o1} = \alpha_1 \; AND$ α_2 and $\alpha_{02} = \alpha_1 \; OR \; \alpha_2$. This operator partially resembles the *geometric crossover operator* [35] proposed by Moraglio et al. We defined such operator specifically for the oracle improvement problem. The merging crossover works well in synergy with our best-match criterion, since merging the 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 if the subtree rooted on this node contains a variable v_i s.t. $\exists s^- \in \mathbb{S}^-$ in which the value of v_i differs from the value of the corresponding state $s^+ \in \mathbb{S}^+$ obtained when executing on the original program the same test that produced s^-. As such, v_i can be used to recognize s^- as a false negative of α, which means that the new assertion could solve such a false negative by predicating on v_i.

Migration. GASSERT periodically exchanges the n best individuals between the two populations,where n is a hyper parameter of the algorithm (see lines 11-14 in 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.

5 Evaluation

To experimentally evaluate our approach, we developed a prototype implementation of GAssert for Java classes, and 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 unguided (random) and invariant-based oracle improvement?*

RQ3 *How does* GAssert *compare with human oracle improvement?*

5.1 Subjects

We conducted our experiments on 34 methods from 7 different Java code bases. We experimented with (i) the *SimpleExamples* (SE) class that contains four small methods that were used in a previous study on oracle improvement [26] to train the humans on the oracle improvement task, (ii) *StackAr* (SA) and *QueueAr* (QA) classes that are publicly released subjects of Daikon often used as evaluation subjects of invariant generators, each of these classes contains five methods, (iii) 20 methods that we selected 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 selected five methods that (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 an assertion oracle. We made no restrictions on the parameter types or on the presence of loops. We selected the last return statements as program point pp.

5.2 RQ1: Effectiveness

To evaluate the effectiveness of GAssert we need an initial set of test cases and some initial assertions to be improved. We also need to perform mutation analysis to get incorrect states, in addition to the correct states obtained by running the test cases on the original program. Then, we run GAssert and get the improved assertions. These are evaluated in terms of false positive and false negative rate on the initial test cases and mutations. To avoid circularity in the evaluation we introduce a new sets of evaluation test cases and mutations. The false positive rate and the mutation score obtained on the latter provide an external assessment of effectiveness.

Initial Correct and Incorrect States. We obtained the initial correct ($\mathbb{S}_0^+$) and incorrect states ($\mathbb{S}_0^-$) by executing an initial test suite on the instrumented version of $\mathcal{P}$ and on a set of initial mutations, respectively. We generated the initial test suite with EvoSuite [13, 15] (v. 1.0.6) and the initial mutations with Major [28] (v. 1.3.4) (line 3 of Algorithm 1). We ran EvoSuite with the branch coverage criterion and a time budget of one minute [13]. We performed ten runs with different random seeds to collect a diverse and large set of initial test cases. We ran Major with all types of supported mutants. Let $\mathcal{T}_0$ and $\mathcal{M}_0$ denote the test cases and mutations produced with EvoSuite and Major, respectively. Columns "$|\mathbb{S}_0^+|$" and "$|\mathbb{S}_0^-|$"of Table 4 show the cardinality of the initial states for each subject. Since GAssert removes redundant states from $\mathbb{S}_0^+$ and $\mathbb{S}_0^-$ and equivalent mutants from $\mathbb{S}_0^-$ with respect to its definition of state equivalence, $|\mathbb{S}_0^-|$ ¡ $|\mathbb{S}_0^+|$ for some subjects.

Initial Assertion Oracles. We obtained an initial assertion for our subjects with Daikon [9] (v. 5.7.2) executed with the initial test suite. We chose Daikon because it represents a fully-fledged and popular tool, often used as the de-facto approach for generating invariants for Java methods [11]. Daikon accepts in input a set of observer methods. We executed Daikon with the same observer methods that GAssert used to obtain primitive variables. Daikon considers all possible exit points of a method, i.e., all returns and exception throw statements. GAssert automatically removes all the initial test cases that do not reach the program point pp, to ensures that Daikon generates invariants that consider only the exit point at pp.

Daikon outputs invariants as a series of precondition α_1, α_2, $\dots\alpha_n$ and postcondition assertions β_1, β_2, $\dots\beta_m$. GAssert converts them into a single (complete) Java assertion in the form of $(\alpha_1$ AND $\alpha_2, \dots$ AND $\alpha_n) \rightarrow$ $(\beta_1$ AND $\beta_2, \dots$ 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.

Evaluation Setup. Table 3 shows the GAssert configuration parameters values used in our experiments. We selected these values with some trial runs according to the basic GP guidelines [2, 52]. We ran GAssert with an overall time budget $\mathcal{B}$ of 90 minutes. To ensure that GAssert will leverage the feedback of OASIs,

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	160	prob. of random node selector	70%

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	$\lvert\mathbb{S}_0^+\rvert$	$\lvert\mathbb{S}_0^-\rvert$	$\lvert\mathcal{T}_V\rvert$	$\lvert\mathcal{M}_V\rvert$	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	1313	31	0	0	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

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 run GASSERT ten times with the same input assertion and correct and incorrect states. We implemented GASSERT to be pseudo-deterministic given the same random seed. We avoid biases in selecting the seeds, by considering 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.

Evaluation Set of Test cases and Mutations. To evaluate if the improved assertions generalize well with unseen test cases and mutations, we obtained a validation set using different tools than the ones used to produce the initial correct and incorrect states for GASSERT: RANDOOP (v. 4.2.0) and PIT (v. 1.4.0) to generate test cases and mutations, respectively. Different tools are expected to generate different test cases and mutations than the ones that GASSERT leveraged during assertion evolution, i.e., those generated by EVOSUITE, MAJOR and OASIS. For each subject, we ran RANDOOP ten times with different random seeds using at least 100 test cases as stopping criterion for each run. Let $\mathcal{T}_V$ and $\mathcal{M}_V$ to denote the test cases and mutations produced by RANDOOP and PIT, respectively. Columns "$|\mathcal{T}_v|$" and "$|\mathcal{M}_v|$" of Table 4 show their cardinality.

Quality Assessment of Assertions. Let α denote the initial assertion and α' the assertion that GASSERT generates. We compute the number of false positives and false negatives of both assertions considering both the initial and evaluation test cases and mutations. We evaluated the assertions on test cases obtained by removing the test oracle assertions generated by EVOSUITE and RANDOOP, and by filtering the test cases that raise exceptions when executed with the original version of the program.

We insert the assertion under evaluation (either α or α') into the method under analysis at the specified $\rho\rho$ for all $\langle test\ cases,\ mutations\rangle$ $\langle\mathcal{T},\mathcal{M}\rangle$ pairs (either $\mathcal{T}_0$ and $\mathcal{M}_0$ or $\mathcal{T}_V$ and $\mathcal{M}_V$). We execute $\mathcal{T}$ and count the number of failing tests, which represents the number of false positives $\mathrm{FP}(\alpha, \mathbb{S}^+)$, *FP* in short. If *FP* is zero, we compute $\mathrm{FN}(\alpha, \mathbb{S}^-)$, *FN* in short, by running mutation testing with $\mathcal{M}$ and $\mathcal{T}$. If *FP* 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 an AND and perform mutation testing with $\mathcal{M}$ and $\mathcal{T}$ for this reduced assertion at $\rho\rho$. If all smaller assertions have false positives then we report FN for *assert(true)*.

While MAJOR returns the source code of each mutation, PIT does not, and thus we cannot compute the number of false negatives for PIT mutation testing but only the mutation score.

Evaluation Results. Table 4 summarizes the evaluation results of RQ1. The subject ID indicates the Java code base. Columns "initial assertion α" indicate the quality of the initial assertion generated with DAIKON. For ten subjects DAIKON does not generate any invariant, and in this case we consider *assert(true)* (with size one) as the initial assertion (column *size*). The false positives on the initial tests (FP_0) are always zero (except for subject CL5), as expected, since DAIKON uses the execution traces of the initial tests to generate α. The size of DAIKON generated assertions ranges from 1 to 147 (28 on average), confirming that DAIKON invariants generate many (often redundant) preconditions and postconditions [11]. The number of false negatives on the initial tests (FN_0) ranges from 0 to 1,750 (209 on average). The number of false positives in the validation tests (FP_v) is always zero, except two subjects, indicating that the initial assertions are successful in generalizing with the unseen tests. This unexpected result may depend on the high branch coverage of the EVOSUITE generated tests. The mutation score of the validation set ranges from 0% to 100% (38% on average).

Columns "GASSERT improves α'" indicate the quality of the GASSERT improved assertions. The table reports the median values of ten execution. The number of iterations ranges from 1 to 9 (3 on average). The median FP of the initial tests (FP_0) is zero for all subjects, as expected, since GASSERT produces only solutions with zero false positives by construction. The median FN of the initial tests (FN_0) ranges from 0 to 1,282 (125 on average). The FN_0 median of α' is less than FN_0 of α by 40% on average, for 30 out of 34 subjects. α' has zero FP_0 and zero FN_0 for 12 subjects, while α has zero FP_0 and zero FN_0 for 4 subjects only. These results demonstrate that GASSERT evolutionary algorithm is effective in improving assertion oracles. α' has more false negatives than α only for the subject SA1. This is due to the fact that 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, while the mutation score of the validation set ranges from 0% to 100% (58% on average). The mutations score of α' is higher than the one of α with the increase of 34% on average, for 16 subjects. When DAIKON produces long assertions GASSERT can reduce their size, showing that GASSERT is able to address all of its three objectives.

5.3 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, described in Algorithm 1. The only difference among GASSERT, RANDOM and INV-BASED is how each of them performs the oracle improvement step (line 6 of Algorithm 1). When running and evaluating RANDOM and DAIKON we used the same evaluation setup that we used for RQ1.

RANDOM is a variant of GASSERT, in which there is no evolutionary pressure in the population because any guidance by the fitness function is disabled. Thus, this variant evolves the two populations in a completely random fashion. To obtain RANDOM we modified GASSERT as follows: (i) we replaced the tournament and best-match parent selection with random selection; (ii) we disabled the Merging crossover and Mutation-based node selector; (iii) we disabled elitism and migration. RANDOM randomly evolves the two populations and terminates when either population finds a perfect solution or the budget expires. In the latter case, RANDOM outputs the best assertion all those produced in the performed generations. We opted to use a random variant of GASSERT rather than another random generator, to perform a fair comparison, where all possibly confounding factors are the same except for the evolutionary pressure. Otherwise, we cannot ensure that differences are due to the search strategy and not due to the differences in implementation details, such as the variables, constants and functions considered.

Column "RANDOM improves α'" shows the quality assessment of the assertion returned by RANDOM. The results show that the improved assertion of GASSERT dominates the one of RANDOM for 20 (59%) and 17

Table 5: Evaluation Results for RQ3

subj.	Initial α			GAssert α'			Human α'						
ID	FP$_v$	$\mathcal{M}_{\text{V}}$	Size	FP$_v$	$\mathcal{M}_{\text{V}}$	Size	type	FP$_v$	$\mathcal{M}_{\text{V}}$	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	0.33	3	M	0	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

(50%) subjects considering the initial and evaluation 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 in 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. 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. Inv-based 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 lead to FP states of α and add it to the test suite.

Column "Daikon improves α'" shows the quality assessment of the assertion returned by Daikon. For ten subjects we do 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.

5.4 RQ3: Comparison with Human Oracle Improvement

Jahangirova et al. [26] 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 GAssert (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 [26], 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. The column *type* in 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. The column *Ov.* shows the overall number of assertions available and the column *Exc.* shows the number of assertions that were excluded.

As the results show, 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. In the column *GPB*, i.e. GAssert performs better, we report the number of manual improvements that achieve a lower mutation score than GAssert does (10 %

of cases).

6 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 generate Boolean expressions (called program invariants) that evaluate to true for all the executions of an input test suite [10, 11, 19, 6, 38, 39, 43]. GASSERT improves oracle assertions by reducing its 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 oracle assertions [6, 36, 47].

Ratcliff et al. evolutionary approach [41] to generate invariants by leveraging negative counterexamples generated with mutation analysis 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.

GASSERT differs from current invariant generators, since it considers both externally (parameters, return values) and internally (local variables, private fields) observable variables. As such, GASSERT assertions are more effective in exposing faults because they can assert about the internal states of methods.

Oracle Quality Metrics. Research on measuring oracle quality mostly focuses on assertions in the test cases (test oracles) [45, 22, 30]. For instance, EVOSUITE [12, 14] and a parameterized test case generator proposed by Fraser and Zeller [16] select from an initial set of possible assertions those that kill the highest number of mutations. These studies propose metrics to measure the quality of oracles in the test cases, to select suitable ones, 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 [55] 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 [24, 25] relies on human input 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 th difficult task of improving assertions with a novel evolutionary approach.

7 Conclusion

Assertion oracles are potent test oracles [3], but designing effective assertion oracles, that is, assertions with no or very few false positives and negatives, is extremely difficult, and automatically generated assertions are still largely imprecise [20, 5, 30].

In this paper, we presented GASSERT, the first automated approach to improve oracle assertions. Our experiments indicate that GASSERT improved assertions present zero false positives, and largely reduce false negatives 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.

References

[1] Angello Astorga, P. Madhusudan, Shambwaditya Saha, Shiyu Wang, and Tao Xie. Learning stateful preconditions modulo a test generator. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '19, pages 775–787. ACM, 2019.

[2] Thomas Back. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford university press, 1996.

[3] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015.

[4] Markus F Brameier and Wolfgang Banzhaf. A comparison with tree-based genetic programming. *Linear Genetic Programming*, pages 173–192, 2007.

[5] Junjie Chen, Yanwei Bai, Dan Hao, Lingming Zhang, Lu Zhang, Bing Xie, and Hong Mei. Supporting oracle construction via static analysis. In David Lo, Sven Apel, and Sarfraz Khurshid, editors, *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 178–189. ACM, 2016.

[6] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. DySy: Dynamic symbolic execution for invariant inference. In *Proceedings of the International Conference on Software Engineering*, ICSE '08, pages 281–290. ACM, 2008.

[7] Jason M. Daida, Adam M. Hilss, David J. Ward, and Stephen L. Long. Visualizing tree structures in genetic programming. In *Genetic and Evolutionary Computation - GECCO 2003, Genetic and Evolutionary Computation Conference*, volume 2724, pages 1652–1664. Springer, 2003.

[8] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[9] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 1999 International Conference on Software Engineering, ICSE' 99, Los Angeles, CA, USA, May 16-22, 1999.*, pages 213–224, 1999.

[10] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[11] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, 2007.

[12] Gordon Fraser and Andrea Arcuri. Evolutionary generation of whole test suites. In *2011 11th International Conference on Quality Software*, pages 31–40. IEEE, 2011.

[13] Gordon Fraser and Andrea Arcuri. 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, pages 416–419. ACM, 2011.

[14] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419. ACM, 2011.

[15] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2013.

[16] Gordon Fraser and Andreas Zeller. Generating parameterized unit tests. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 364–374. ACM, 2011.

[17] Juan P. Galeotti, Carlo A. Furia, Eva May, Gordon Fraser, and Andreas Zeller. Inferring loop invariants by mutation, dynamic analysis, and static checking. *IEEE Trans. Software Eng.*, 41(10):1019–1037, 2015.

[18] Juan Pablo Galeotti, Carlo A. Furia, Eva May, Gordon Fraser, and Andreas Zeller. Dynamate: Dynamically inferring loop invariants for automatic full functional verification. In Eran Yahav, editor, *Hardware and Software: Verification and Testing - 10th International Haifa Verification Conference, HVC 2014, Haifa, Israel, November 18-20, 2014. Proceedings*, volume 8855 of *Lecture Notes in Computer Science*, pages 48–53. Springer, 2014.

[19] Ashutosh Gupta and Andrey Rybalchenko. Invgen: An efficient invariant generator. In *International Conference on Computer Aided Verification*, CAV '09, pages 634–640, 2009.

[20] Mark Harman, Sung Gon Kim, Kiran Lakhotia, Phil McMinn, and Shin Yoo. Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem. In *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010, Workshops Proceedings*, pages 182–191. IEEE Computer Society, 2010.

[21] Mark Harman, William B. Langdon, Yue Jia, David Robert White, Andrea Arcuri, and John A. Clark. The GISMOE challenge: constructing the pareto program surface using genetic programming to find better programs (keynote paper). In Michael Goedicke, Tim Menzies, and Motoshi Saeki, editors, *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*, pages 1–14. ACM, 2012.

[22] Chen Huo and James Clause. Improving oracle quality by detecting brittle assertions and unused inputs in tests. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 621–631, New York, NY, USA, 2014. ACM.

[23] Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella. Test oracle assessment and improvement. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 247–258. ACM.

[24] Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella. Test oracle assessment and improvement. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 247–258. ACM, 2016.

[25] Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella. OASIs: Oracle Assessment and Improvement Tool. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 368–371. ACM, 2018.

[26] Gunel Jahangirova, David Clark, Mark Harman, and Paolo Tonella. An empirical validation of oracle improvement. *IEEE Transactions on Software Engineering*, 2019.

[27] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, September 2011.

[28] René Just. The major mutation framework: Efficient and scalable mutation analysis for java. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA '14, pages 433–436. ACM, 2014.

[29] John R Koza and John R Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

[30] William B. Langdon, Shin Yoo, and Mark Harman. Inferring automatic test oracles. In *10th IEEE/ACM International Workshop on Search-Based Software Testing, SBST@ICSE 2017, Buenos Aires, Argentina, May 22-23, 2017*, pages 5–6. IEEE, 2017.

[31] Y. Lavinas, C. Aranha, T. Sakurai, and M. Ladeira. Experimental analysis of the tournament size on genetic algorithms. In *2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 3647–3653, 2018.

[32] David Lo and Shahar Maoz. Mining scenario-based specifications with value-based invariants. In Shail Arora and Gary T. Leavens, editors, *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, pages 755–756. ACM, 2009.

[33] Phil McMinn, David W. Binkley, and Mark Harman. Empirical evaluation of a nesting testability transformation for evolutionary testing. *ACM Trans. Softw. Eng. Methodol.*, 18(3):11:1–11:27, 2009.

[34] Brad L Miller, David E Goldberg, et al. Genetic algorithms, tournament selection, and the effects of noise. *Complex systems*, 9(3):193–212, 1995.

[35] Alberto Moraglio, Krzysztof Krawiec, and Colin G Johnson. Geometric semantic genetic programming. In *International Conference on Parallel Problem Solving from Nature*, pages 21–31. Springer, 2012.

[36] Cu D Nguyen, Alessandro Marchetto, and Paolo Tonella. Automated oracles: An empirical study on cost and effectiveness. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, FSE '13, pages 136–146. ACM, 2013.

[37] Annibale Panichella, Rocco Oliveto, Massimiliano Di Penta, and Andrea De Lucia. Improving multi-objective test case selection by injecting diversity in genetic algorithms. *IEEE Trans. Software Eng.*, 41(4):358–383, 2015.

[38] Corina S. Pasareanu and Willem Visser. Verification of java programs using symbolic execution and invariant generation. In Susanne Graf and Laurent Mounier, editors, *Model Checking Software, 11th International SPIN Workshop, Barcelona, Spain, April 1-3, 2004, Proceedings*, volume 2989 of *Lecture Notes in Computer Science*, pages 164–181. Springer, 2004.

[39] Long H. Pham, Jun Sun, Lyly Tran Thi, Jingyi Wang, and Xin Peng. Learning likely invariants to explain why a program fails. In *22nd International Conference on Engineering of Complex Computer Systems, ICECCS 2017, Fukuoka, Japan, November 5-8, 2017*, pages 70–79. IEEE Computer Society, 2017.

[40] Dipesh Pradhan, Shuai Wang, Shaukat Ali, Tao Yue, and Marius Liaaen. CBGA-ES: A cluster-based genetic algorithm with elitist selection for supporting multi-objective test optimization. In *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*, pages 367–378. IEEE Computer Society, 2017.

[41] Sam Ratcliff, David R. White, and John A. Clark. Searching for invariants using genetic programming and mutation testing. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11, pages 1907–1914, New York, NY, USA, 2011. ACM.

[42] Henry G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[43] Abhik Roychoudhury and I. V. Ramakrishnan. Inductively verifying invariant properties of parameterized systems. *Autom. Softw. Eng.*, 11(2):101–139, 2004.

[44] Federica Sarro, Filomena Ferrucci, Mark Harman, Alessandra Manna, and Jian Ren. Adaptive multi-objective evolutionary algorithms for overtime planning in software projects. *IEEE Trans. Software Eng.*, 43(10):898–917, 2017.

[45] D. Schuler and A. Zeller. Assessing oracle quality with checked coverage. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pages 90–99, March 2011.

[46] Oren Shoval, Hila Sheftel, Guy Shinar, Yuval Hart, Omer Ramote, Avi Mayo, Erez Dekel, Kathryn Kavanagh, and Uri Alon. Evolutionary trade-offs, pareto optimality, and the geometry of phenotype space. *Science*, 336(6085):1157–1160, 2012.

[47] Matt Staats, Shin Hong, Moonzoo Kim, and Gregg Rothermel. Understanding user understanding: Determining correctness of generated program invariants. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA '12, pages 188–198, New York, NY, USA, 2012. ACM.

[48] Hisashi Tamaki, Hajime Kita, and Shigenobu Kobayashi. Multi-objective optimization by genetic algorithms: A review. In *Proceedings of IEEE international conference on evolutionary computation*, pages 517–522. IEEE, 1996.

[49] Tao Xie, D. Notkin, and D. Marinov. Rostra: a framework for detecting redundant object-oriented unit tests. In *Proceedings. 19th International Conference on Automated Software Engineering, 2004.*, pages 196–205, 2004.

[50] Willem Visser, Corina S. Pasareanu, and Sarfraz Khurshid. Test input generation with java pathfinder. In George S. Avrunin and Gregg Rothermel, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004, Boston, Massachusetts, USA, July 11-14, 2004*, pages 97–107. ACM, 2004.

[51] Shuai Wang, Shaukat Ali, Tao Yue, and Marius Liaaen. Integrating weight assignment strategies with NSGA-II for supporting user preference multiobjective optimization. *IEEE Trans. Evolutionary Computation*, 22(3):378–393, 2018.

[52] Darrell Whitley. A genetic algorithm tutorial. *Statistics and computing*, 4(2):65–85, 1994.

[53] Lingming Zhang, Tao Xie, Lu Zhang, Nikolai Tillmann, Jonathan de Halleux, and Hong Mei. Test generation via dynamic symbolic execution for mutation testing. In *26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania*, pages 1–10. IEEE Computer Society, 2010.

[54] Lingming Zhang, Guowei Yang, Neha Rungta, Suzette Person, and Sarfraz Khurshid. Feedback-driven dynamic invariant discovery. In Corina S. Pasareanu and Darko Marinov, editors, *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, pages 362–372. ACM, 2014.

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