GAssert: A Fully Automated Tool to Improve Assertion Oracles

Valerio Terragni*, Gunel Jahangirova†, Paolo Tonella† and Mauro Pezzè†‡
*University of Auckland, Auckland, New Zealand
†Università della Svizzera italiana, Lugano, Switzerland
‡Schaffhausen Institute of Technology, Schaffhausen, Switzerland
v.terragni@auckland.ac.nz {gunel.jahangirova, paolo.tonella, mauro.pezze}@usi.ch

This is the author's version of the work. The definitive version appeared at ICSE 2021

Abstract—This demo presents the implementation and usage details of GASSERT, the first tool to automatically improve assertion oracles. 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 are prone to both false positives (the assertion fails but should pass) and false negatives (the assertion passes but should fail). Given a Java method containing an assertion oracle to improve, GASSERT returns an improved assertion with fewer false positives and false negatives than the initial assertion. Internally, GASSERT implements a novel co-evolutionary algorithm that explores the space of possible assertions guided by two fitness functions that reward assertions with fewer false positives, fewer false negatives, and smaller size.

Index Terms—oracle improvement, program assertions, the oracle problem, evolutionary algorithm, genetic programming, automated test generation, mutation analysis

I. INTRODUCTION

Assertion oracles (also called program assertions) are executable boolean expressions placed inside the program 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 [1]. For most non-trivial programs, designing perfect oracles is difficult, and thus assertion oracles often fail to distinguish between correct and incorrect executions [2], that is, they are prone to both false positives and false negatives. A *false positive* is a correct program state in which the assertion fails (but should pass). 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* [2].

Improving the quality of assertion oracles by removing their deficiencies is an important problem and would bring several benefits, primarily the reduced false alarm rate and increased fault detection capability of test suites. Notably, automated test case generators [3], [4] will benefit the most from improved assertion oracles. This is because high quality assertion oracles avoid the need to automatically define a test oracle for each generated test case. Indeed, the *oracle problem* [1] is a major obstacle in test automation, limiting the effectiveness of automatically generated test suites [3], [4].

Recently, Jahangirova et al. proposed the OASIs [2] approach to automatically generate test cases and mutations that expose the oracle deficiencies of a given assertion oracle. The OASIs's output is intended to help developers improve the oracles. However, even with the oracle deficiencies provided by OASIS, manually improving assertion oracles remains difficult [5]. In fact, the authors of OASIs report that for only 67% of the given assertions humans successfully removed all oracle deficiencies reported by OASIs [5].

GASSERT [6] (*Genetic ASSERTion improvement*) is the first technique to automatically improve assertion oracles. The envisioned users of GASSERT are JAVA developers who wish to improve the quality of their assertion oracles. Given an assertion oracle α and its oracle deficiencies provided by OASIs, GASSERT explores the space of possible assertions to return a new assertion α' with zero false positives and the smallest number of false negatives. GASSERT favors assertions with zero false positives, as false alarms trigger an expensive debugging process.

Internally, GASSERT implements a novel co-evolutionary algorithm that evolves two populations of assertions in parallel 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 first population rewards solutions with fewer false positives, the second population those with fewer false negatives, considering the remaining objectives only in tie cases. On a regular basis, the two populations exchange their best individuals (population migration) to supply the other population with good genetic material useful to improve the secondary objective.

We evaluated the ability of GASSERT to improve an initial set of DAIKON [7] generated assertions on 34 methods from 7 JAVA code bases [6]. The improved assertions have zero false positives and reduce the false negatives by 40% (on average) with respect to the initial DAIKON assertions. When evaluated with unseen tests and mutants, the assertions generated by GASSERT increase the mutation score by 34% (on average).

This paper extends our recent ESEC/FSE paper [6] by giving details about the design, implementation and usage of GASSERT, which can be found at:
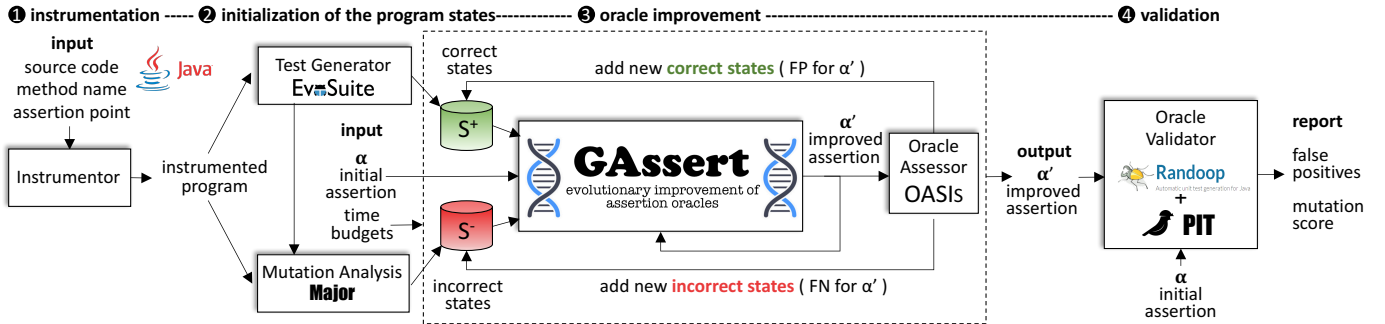
`https://github.com/valerio-terragni/gassert`

Figure 1. Logical workflow of GASSERT

II. GASSERT

GASSERT is a command-line tool, which has five inputs: (i) the source code of a JAVA class, (ii) the name of the method under analysis, (iii) an initial assertion oracle α, (iv) the point in the method were α is placed (assertion point), and (v) a global and an internal time budget for the oracle improvement. The output of GASSERT is an improved assertion oracle α'. Figure 1 shows the logical workflow of GASSERT, which is composed of four phases:

❶ **instrumentation**, GASSERT instruments the method under analysis to capture program states at runtime.

❷ **initialization of the program states**, GASSERT produces an initial set of correct and incorrect states ($\mathcal{S}^+$ and $\mathcal{S}^-$) by executing an initial test suite on the instrumented version of the original method and on its faulty variants (mutations).

❸ **oracle improvement**, this phase alternatively executes GASSERT and OASIS until a time budget expires or OASIS does not find oracle deficiencies for the improved oracle α'.

❹ **validation**, GASSERT evaluates the initial and improved assertions (α and α') on a validation set of tests and mutations.

Figure 2 shows a running example of GASSERT applied to method floor of class FastMath of The Apache Commons Math library. The method implements a fast algorithm for the floor computation, which takes as input a real number x, and outputs the greatest integer less than or equal to x (result).

Figure 2 (a) shows the instrumented version of the method and the initial assertion α : (y == result) && (x > result) (line 16). Although α properly behaves in many correct and incorrect states, it has both false positives and false negatives.

Figure 2 (b) shows an example of a false positive program state for α, together with the EVOSUITE test case that produces it. For this correct state, x is not greater than result, and thus α returns false (but should return true).

Figure 2 (c) shows an example of a false negative program state for α, together with the EVOSUITE test case and MAJOR mutation that produce it. For this incorrect state, the values of y and result are wrong (they should be 0), but α does not fail (it returns true instead of false).

Running GASSERT provides an improved assertion α': (y == result) && (x >= result) && (x < (result+1)), which intuitively captures the behavior of a *floor* function, and it does not suffer from the oracle deficiencies of Figure 2 (b) and (c). The following subsections describe the four phases in detail.

A. Instrumentation

The *Instrumentor* component of GASSERT inserts additional method calls in the method under analysis to collect program states at runtime. We implemented it by relying on the source code manipulator JAVAPARSER (v. 3.6.26)[1].

The instrumentor analyzes the source code of the method under analysis to collect the method parameters (including the object receiver) *MP*, and all the local variables *LV* that are visible at the assertion point. For the example in Figure 2, $MP = \{x\}$ and $LV = \{y, result\}$. It then instruments two method calls. The first one is placed at the beginning of the method passing *MP* as an argument (line 3 in Figure 2). The second one is located right before the assertion point passing both *MP* and *LV* as arguments (line 15 in Figure 2). By considering the parameter values at the beginning of the method, GASSERT is able to generate assertions that predicate on method preconditions. GASSERT distinguishes the variables in *MP* at the two execution points by adding the prefix "old_" to the variable names at the first execution point.

B. Initialization of the Program States

GASSERT needs test cases and mutations to initialize the repositories of correct and incorrect states ($\mathcal{S}^+$ and $\mathcal{S}^-$). Such repositories are progressively filled with the correct and incorrect states returned by OASIS. The rationale for initializing the repositories, instead of immediately relying on states returned by OASIS, is to minimize the number of iterations. In this way, we avoid using OASIS to detect obvious oracle deficiencies, and rather let OASIS focus on hard-to-find ones.

To enable full automation, GASSERT obtains the initial test cases and mutations from EVOSUITE [4] (v. 1.0.6)[2] and MAJOR (v. 1.3.4)[3], respectively. EVOSUITE is a search-based test case generator for JAVA driven by various coverage criteria. In our experiments we used the branch coverage criterion. MAJOR generates source-code mutants of a JAVA class by seeding artificial faults. Notably, developers and testers can also provide additional manually-written test cases and faulty versions of the method under analysis, which likely yield to assertion oracles of higher quality. Figure 2 (b) and (c) show examples of test cases and mutations generated at phase ❷.

[1]http://javaparser.org/
[2]https://www.evosuite.org/
[3]https://mutation-testing.org/

```java
1  public class FastMath{
2    public static double floor(final double x) {
3      ch.usi.gassert.visitor.GAVisitor.getInstance()
         .enterMethod("floor", "old_x", x);
4      // [..] omitted code for brevity
5      long y = (long) x;
6      if (x < 0 && y != x) {
7        y--;
8      }
9      final double result;
10     if (y == 0) {
11       result = x * y;
12     } else {
13       result = y;
14     }
15     ch.usi.gassert.visitor.GAVisitor.getInstance()
         .assertionPoint("floor", "x", x, "y", y, "result", result);
16     assert ((y == result) && (x > result));
17     return result } }
```
(a)

False Positive

Ev**S**uite

```java
@Test(timeout = 4000)
public void test1 () throws Throwable {
  double double0 = FastMath.floor(5.0);
}
```

```json
{ "versionId": "original",
  "testId": "test1",
  "methodName": "floor",
  "identifier2value": {
    "old_x": 5.0,
    "x": 5.0,          program state
    "y": 5,             {JSON}
    "result": 5.0,
} }
```
(b)

False Negative

🐞 Major

if (x < **1** && y != x) { (line 6)

Ev**S**uite

```java
@Test(timeout = 4000)
public void test2 () throws Throwable {
  double double0 = FastMath.floor(0.33);
}
```

```json
{ "versionId": "mutants/18/",
  "testId": "test1",
  "methodName": "floor",
  "identifier2value": {
    "old_x": 0.33,
    "x": 0.33,          program state
    "y": -1, 🐞           {JSON}
    "result": -1.0, 🐞
} }
```
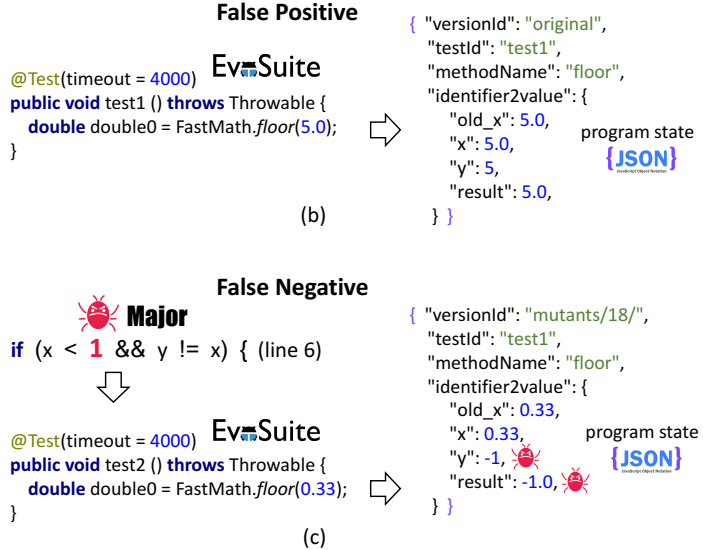(c)

Figure 2. Instrumented version of the method floor (a), examples of a false positive (b) and a false negative (c) for the assertion at line 17.

GASSERT initializes $\mathcal{S}^+$ by executing the test cases on the instrumented method, and $\mathcal{S}^-$ by executing the test cases on the instrumented mutations. Invoking the instrumented code constructs the program states and saves them on disk as JSON files. Saving the states is crucial to quickly calculate how many false positives and false negatives a candidate assertion has, without requiring expensive program re-executions. Indeed, our evolutionary algorithm might explore millions of candidate assertions before converging to an optimized solution.

GASSERT generates assertion oracles as Boolean expressions that predicate on variables and functions of Boolean or numerical types (see Table I). As such, a program state s is a set of Boolean or numerical variables $\{v_1, \cdots, v_k\}$. Each variable v_i has a type, an identifier, and a value. For each variable v_i passed as arguments to the instrumented method calls, GASSERT constructs a program state s as follows:

If v_i is of primitive type (Boolean or numerical), GASSERT simply adds its runtime value to s (rounding floats with a fixed precision) using the variable name as identifier. If v_i is non-primitive (objects in JAVA), GASSERT needs to convert v_i into primitive values. GASSERT achieves this with a *hybrid state serialization* that combines the object serialization and observer abstraction approaches [6]. *Object serialization* captures the values of primitive-type object fields that are recursively reachable from v_i. *Observers abstraction* 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.

GASSERT automatically finds the observer methods of v_i's class with an efficient (but conservative) byte-code static analyzer. The analyzer marks a method as side-effect free if it cannot directly or indirectly execute putfield or putstatic bytecode instructions. GASSERT relies on JAVA REFLECTION to get at runtime the values of primitive fields and the return values of the observer methods invocations. Figure 2 (b) and (c) show the JSON files of the program states obtained when executing the corresponding test cases.

Table I
FUNCTIONS 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

C. Oracle Improvement

The oracle improvement process takes in input an initial assertion α and two time budgets (an internal one for the co-evolutionary algorithm and a global one for the whole process), and outputs an improved assertion α'. The initial assertion can be specified by the user or automatically generated by our scripts using the invariant generator DAIKON (v. 5.7.2) [7]. The default configuration is an internal time budget of 30 minutes and a global one of 90 minutes.

The oracle improvement process is composed of three steps:
I. GASSERT executes the co-evolutionary algorithm and terminates when it finds an assertion α' with zero oracle deficiencies with respect to the current correct and incorrect states ($\mathcal{S}^+$ and $\mathcal{S}^-$) or the internal time budget expires. In the latter case, GASSERT returns the assertion α' that among all the explored assertions with zero false positives has the lowest number of false negatives.
II. OASIS searches for oracle deficiencies of α'. If it finds them, it adds the resulting program states to $\mathcal{S}^+$ and $\mathcal{S}^-$.
III. GASSERT takes in input α' and repeats step I.

Such an iterative process terminates when OASIS does not find oracle deficiencies for α' or the global time budget expires.

The co-evolutionary algorithm evolves two populations of assertions in parallel. One population uses the number of false positives as the fitness score, while the other uses the number of false negatives. The remaining objectives are used only in tie cases. The two populations periodically exchange their best individuals to help optimize the secondary objectives.

GASSERT evolves each population following the GP approach: (i) **selection**: selecting pairs of assertions (parents) by means of fitness functions that reward fitter solutions; (ii) **crossover**: creating new (and possibly fitter) offspring by combining portions of the parent assertions; and, (iii) **mutation**: mutating the offspring (with a certain probability). GASSERT adopts a tree-like representation of assertions and uses the standard tree-based mutation and crossover operators. Moreover, we propose novel selection and crossover operators that are specific to the oracle improvement problem [6].

We now exemplify how our algorithm could obtain α' : `(y == result) && (x >= result) && (x < (result+1))` in the example of Figure 2. Let us assume that the algorithm selects two parents α_{p1} : `(y == result) && (x > result)` and α_{p2} : `(x < (result+1))`. If crossover produced `(y == result) && (x > result) && (x < (result+1))` and mutation changed `>` to `>=`, GASSERT would obtain the improved assertion α'.

OASIS [2] detects false positives of an assertion α by creating a new branch with the negated boolean expression of α. It then uses search-based test generation [4] to produce test cases that cover the branch and consequently make the assertion fail. For instance, given the assertion at line 17 in Figure 2 (a), OASIS would obtain the test case in Figure 2 (b) by replacing the assertion with the artificial branch `if((y != result) || (x <= result)){}` and then driving search-based test generation towards covering this branch.

OASIS detects false negatives by combining test case generation and mutation testing. It injects faults into the program and generates test cases for the faulty version such that at least one of the variables used in the assertion changes its value, while the outcome of the assertion does not change.

D. Validation

To evaluate if the improved assertions generalize well with unseen correct and incorrect states, we generated new test cases and mutations using RANDOOP [3] (v. 4.2.0)[4] and PIT (v. 1.4.0)[5], respectively. Because they are different tools from the ones that provide test cases and mutations to the oracle improvement process (EVOSUITE, MAJOR and OASIS), they are expected to provide different test cases and mutations.

The validation phase counts the number of validation tests that fail with the improved assertion inserted at the assertion point. If it is zero, we use PIT to run the validation tests and report the mutation score. If it is greater than zero, we cannot run PIT because we need a green test suite. In such case, if the evaluated assertion has the form `assert($\alpha_1$ && $\alpha_2$ && $\alpha_3$)`, GASSERT considers each of the smaller assertions `assert($\alpha_1$)`, `assert($\alpha_2$)` and `assert($\alpha_3$)` removing those that have false positives. Then, it concatenates the remaining smaller conditions with `&&` and it performs mutation testing with PIT for this reduced assertion. It then repeats this process for the initial assertion. The user of GASSERT can compare the HTML reports of PIT to better understand the fault detection capability of the initial and improved assertion oracles.

---

[4]https://randoop.github.io/randoop/
[5]https://pitest.org/

III. EVALUATION

We evaluated GASSERT on 34 methods from 7 different Java code bases [6]. The validation phase shows that improved assertions have always zero false positives and achieve, on average, 34% increase in mutation score with respect to initial DAIKON assertions.

In addition, we compared GASSERT to two baselines RANDOM and INV-BASED. RANDOM is a variant of GASSERT with no guidance by the fitness functions. The results show that GASSERT-generated assertions outperform the RANDOM-generated ones for 50% of the subjects. INV-BASED executes DAIKON on the initial test suite and obtains an invariant. It then augments the initial test suite with the test cases generated by OASIS that reveal false positives, and re-executes DAIKON. This process repeats until the global time budget expires. GASSERT outperforms INV-BASED for 63% of the subjects.

We also compared GASSERT with a set of human-improved assertions collected from 102 developers [5]. Our results show that the mutation score achieved by GASSERT is always equal to the average mutation score achieved by the humans. Moreover, 10% of the human-improved assertions achieve a lower mutation score than the assertions improved by GASSERT.

IV. CONCLUSION

While there are many techniques to automatically generate program assertions (e.g., program invariants [7]), automatically improving assertion oracles by removing false positives and false negatives is an unexplored problem. GASSERT is the first technique of its kind, opening a new research area.

Techniques like GASSERT might encourage developers to use assertion oracles more often, resulting in better software quality in the long run. We highlight three promising future research directions: (i) leverage the feedback of OASIS not only after GASSERT has produced the final assertion, but also during the evolution, (ii) increase the expressiveness of the assertions (e.g., with universal quantifiers), and (iii) make the improved assertions easier to read and understand for humans.

REFERENCES

[1] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, 2015.
[2] G. Jahangirova, D. Clark, M. Harman, and P. Tonella, "Test Oracle Assessment and Improvement," in *Proceedings of the International Symposium on Software Testing and Analysis*, ser. ISSTA '16. ACM, 2016, pp. 247–258.
[3] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proceedings of the International Conference on Software Engineering*, ser. ICSE '07. ACM, 2007, pp. 75–84.
[4] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2013.
[5] G. Jahangirova, D. Clark, M. Harman, and P. Tonella, "An empirical validation of oracle improvement," *IEEE Transactions on Software Engineering*, 2019.
[6] V. Terragni, G. Jahangirova, P. Tonella, and M. Pezzè, "Evolutionary improvement of assertion oracles," in *Proceedings of the Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE '20. ACM, 2020, pp. 1178–1189.
[7] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," in *Proceedings of the International Conference on Software Engineering*, ser. ICSE '99. ACM, 1999, pp. 213–224.