

SmallEvoTest: Genetically Created Unit Tests for Smalltalk

Alexandre Bergel¹, Geraldine Galindo-Gutiérrez², Alison Fernandez-Blanco³ and Juan-Pablo Sandoval-Alcocer³

¹*RelationalAI, Switzerland*

²*CICEI, Universidad Católica Boliviana "San Pablo"*

³*Department of Computer Science, School of Engineering, Pontificia Universidad Católica de Chile*

Abstract

Evolutionary test generation techniques have emerged as a popular approach in recent years for enhancing the testing of software systems. However, while these techniques have proven to be efficient in programming languages that support static type annotations, dynamically typed programming languages have not received significant attention from the automatic test generation community.

This paper introduces an approach aimed at automatically generating fully executable unit tests suitable for dynamically typed programming languages. In particular, our approach is tuned for dynamically-typed and class-based programming languages, and it is implemented in the Pharo and GToolkit programming languages. To address the absence of static type annotations, our approach uses a type profiling mechanism and employs a genetic algorithm to drive the evolution of the unit tests.

Keywords

Automatically Test Suite Generation, Genetic Algorithms, Pharo Programming Language

1. Introduction

Automatically Test Suite Generation (ATSG) consists in creating executable unit tests for a particular class. ATSG produces unit tests that exercise methods for a given target class. Such generated tests complement manually hand-written tests by (i) focussing on untested branches or code portions or (ii) exercising corner-case scenarios. ATSG has been gaining popularity thanks to EvoSuite¹ and Randoop² for Java [1, 2, 3, 4].


EvoSuite considers the automatic test generation as a mathematical optimization process through an evolutionary algorithm and a fitness function [5, 6]. In particular, the evolution of the unit tests being generated is designed to maximize the branch coverage of the class being tested [7].


This short paper presents *SmallEvoTest*, a tool for Pharo to create unit tests for a particular class automatically. Similarly to EvoSuite and Randoop, SmallEvoTest does not require a training

IWST 2023: International Workshop on Smalltalk Technologies, August 29-31, 2023, Lyon, France

 <https://bergel.eu> (A. Bergel)

 0000-0002-0877-7063 (A. Bergel); 0009-0002-3801-5227 (G. Galindo-Gutiérrez); 0000-0003-1784-814X (A. Fernandez-Blanco); 0000-0002-8335-4351 (J. Sandoval-Alcocer)

 © 2023 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

¹<https://www.evosuite.org/>

²<https://randoop.github.io/randoop/>

dataset and does not use any large language model such as ChatGPT. SmallEvoTest is available online³ under the MIT license.

Outline. The paper is organized as follows. Section 2 gives a running example of SmallEvoTest; Section 3 gives a highlight of a number of design aspects of our tool; Section 4 lists studies and tools related to this paper. Section 5 concludes and outlines our future work.

2. SmallEvoTest In A Nutshell

SmallEvoTest is relatively easy to configure and use. To illustrate this, let's consider the class `GCPoint`, which is defined as follows:

```
Object subclass: #GCPoint
  instanceVariableNames: 'x y'

GCPoint>>initialize
  super initialize.
  x := 0.
  y := 0

GCPoint>>add: anotherPoint
  ↑ GCPoint new x: x + anotherPoint x y: y + anotherPoint y; yourself

GCPoint>>negated
  ↑ GCPoint new x: x negated y: y negated; yourself

GCPoint>>x: xValue y: yValue
  x := xValue.
  y := yValue.

GCPoint>>x
  ↑ x

GCPoint>>y
  ↑ y
```

This simple class mimics the standard `Point` class, and we use this example throughout this paper. Using SmallEvoTest, unit tests for this class can be generated by executing the following code:

```
SmallEvoTest new
  targetClass: GCPoint;
  generateTestNamed: #GCPointTest;
  numberOfTestsToBeCreated: 15;
  nbOfStatements: 8;
  executionScenario: [
    (GCPoint new x: 3 y: 10)
      add: (GCPoint new x: 1 y: 12) ];
  run.
```

The class `SmallEvoTest` expects as arguments the target class (the `GCPoint` class for which we want to generate unit tests), the name of the test case to be generated (`GCPointTest`), and an execution scenario block exercising the target class. The execution scenario block is meant to

³<https://github.com/bergel/GeneticallyCreatedTests>

provide hints about the argument types. In this example, the scenario block invokes `x:y:` and `add:` with some arguments. Note that the result of the scenario is not used.

As a result, the class `GCTest` is created and will contain 15 test methods, each with 8 statements (excluding the assertions). Here is an example of how a test method looks like:

```
GCTest>>testGENERATED10
| v1 v2 v3 v4 v5 v6 v7 v8 |
v1 := GCTest new.
v2 := 4.
v3 := v1 x: v2 y: v2 .
v4 := v3 negated.
v5 := GCTest new.
v6 := v1 y.
v7 := v3 negated.
v8 := v5 add: v3 .
self assert: v4 printString equals: 'GCTest(-4,-4)'.
self assert: v6 equals: (4).
self assert: v7 printString equals: 'GCTest(-4,-4)'.
self assert: v8 printString equals: 'GCTest(4,4)'.
```

This test has been produced by genetic algorithms and the generation process was guided by the objective to maximize the number of executed methods of the target class. Each of the twenty generated test methods has eight statements (indicated with the assignments of `v1` to `v8`) and a number of assertions. Note that `SmallEvoTest` uses a set of hyperparameters, including the number of tests to be generated and the number of (non-assertion) statements to be contained in a test.

As the invocation of `SmallEvoTest` illustrates, three essential parameters must be provided to generate tests. First, the class to be tested is specified using `targetClass:`. Generated tests will directly exercise the methods defined in this class. The result of the test generation will be kept as test methods in a class named `GCTest`. The code provided as a block to `executionScenario:` is meant to exercise the class under test and is solely used to extract argument types.

A central aspect of `SmallEvoTest` is to use a type profiling technique to infer possible types to be provided. In our example above, the scenario invokes (i) `x:y:` with two integers and (ii) `add:` with another point. Type information is useful to produce and use object examples during the test generation. Such examples are used to provide the necessary type information when generating and composing statements through genetic operations. In particular:

- The fact that `x:y:` uses two integers lead to the creation of the statement `v2`, then `v3` is produced, and
- `add:` use another point as a parameter, it produces the statement `v8`, using `v3` as argument.

The next section describes some of the design aspects we had to consider when generating unit tests for `Smalltalk`.

3. Design of `SmallEvoTest`

3.1. Background: Genetic Algorithms

Genetic algorithms are a type of machine learning algorithm inspired by biological and natural evolution principles. In genetic algorithms, a *population* is made of *individuals*, and each

individual has a *chromosome*. A chromosome is a linear sequence of *values*. Genetic algorithms are commonly employed to *mathematically optimize* a function $f(x) = y$, i.e., finding a sequence of x leading to maximize the value y . The variable x is a datapoint in a multi-dimensional domain, and the variable y is a number. The function f is called *fitness function*, which indicates how fit the individual x is. The function f may model an arbitrary complex operation, such as generating unit tests produced (obtained from the variable x) and measuring the coverage of the target class (the y variable).

Evolution with genetic algorithms happens by randomly selecting fit individuals from a given population, breeding these individuals through genetic operations to produce a new population. The population becomes fitter with each generation by producing individuals with higher fitness values. Overall, the population is getting fitter, thus increasing the likelihood of finding the optimal solution (i.e., the highest possible value of y).

3.2. Genetic Encoding

Applying genetic algorithms to produce a test implies that the content of the test must be adequately encoded as a chromosome. We denote x as a test and refer to the number of covered methods of the target class with y . By optimizing $f(x) = y$, genetic algorithms will search for a test x with high code coverage.

The test `testGENERATED10` given above consists of two parts: (i) initialization of the tests made of object creations and message sends, and (ii) assertions. As produced by SmallEvoTest, assertions do not contribute to increasing the test coverage; as such, we exclude the assertion generation from the genetic encoding to treat it in a separate way⁴.

The unit test x is a value in the space S^N where S corresponds to the domain of statements, and N is the length of the test to be generated in terms of a number of statements. We consider two kinds of statements, either an *object creation* or a *message send*. For example, if we arbitrarily say that $N = 8$ (i.e., generated test method will have eight statements as in the example above), then x will be encoded as (s_1, s_2, \dots, s_8) , in which each s_i is either an object creation or a message send. In the example given above, s_1 corresponds to the statement `v1 := GCPPoint new`, an object creation, while s_3, s_4, s_6, s_7, s_8 corresponds to message sends.

3.3. Mutation and crossover

Genetic algorithms employ two biologically-inspired operators, *mutation* and *crossover*. A mutation consists in replacing a statement with another. As such, a message sent can be replaced by sending another message, e.g., `v4 := v3 negated` is replaced by `v4 := v3 add: v1` or by an object creation. A crossover replaces a segment in an individual with a segment from another individual. Consider two tests $x = \{s_1, s_2, s_3, s_4, s_5\}$ and $x' = \{s'_1, s'_2, s'_3, s'_4, s'_5\}$, a possible result for $crossover(x, x') = \{s_1, s_2, s'_3, s'_4, s'_5\}$, assuming a cutpoint on the third statement.

After each genetic operation, variables used as message arguments may have to be readjusted to satisfy type requirements. For example, if s'_3 was originally `v3 := v2 add: v1` then it expects `v2` and `v1` to be a `GCPPoint`. However, in the result of the crossover, `v1` (defined in s_1) and `v2`

⁴Note that this decision was also taken in EvoSuite.

(defined in s_2) may have different types. The receiver and arguments of a message may have to be replaced by variables meeting the type requirements.

3.4. Generating Assertions

During the source code generation, assertions are appended to the statement source code. The test's statements are executed in a local environment, and assertions are produced by determining simple equality relations against different variables.

In the current version of SmallEvoTest, assertions are produced for leaf variables, i.e., not used as argument or receiver or other statements.

4. Related Work

In recent years, ATSG has gained popularity with the introduction of new or improved generation tools [8, 9]. An example of this growth can be seen in the SBST Tool Contest, which reached its 10th edition in 2022 in the category of Java Unit Testing Contest. Two of its participants, EvoSuite and Randoop, have been awarded for several years [10, 11].

EvoSuite. Using a genetic algorithm to produce unit tests was pioneered by EvoSuite⁵ [5]. EvoSuite evolves unit tests in a similar fashion as we do and operates for the Java programming language. SmallEvoTest uses some of the ideas from EvoSuite, such as test evolution, individual encoding, and test generation. However, SmallEvoTest provides an explicit repository for type information populated with a code example.

Randoop. A popular alternative to EvoSuite is Randoop⁶, which is a Java unit test generator that uses feedback-directed random generation which consists of creating statements using a randomly chosen method call and previous statements as arguments [13]. The result of executing each new statement is then verified by the tool. EvoSuite uses evolutionary search to generate test suites [5]. It is guided by multiple coverage criteria (e.g., branch distance, mutation testing) [14]. Studies have shown that its latest search algorithm, DynaMOSA (Dynamic Many-Objective Sorting Algorithm) [15, 16, 17], produces short tests with higher coverage than previous algorithms (e.g., MOSA [18], WSA [19]).

Pynguin. Besides test generation in Java language, Lukasczyk et al. recently presented Pynguin (Python General Unit Test Generator) [20, 21]. This tool uses evolutionary algorithms to explore the challenge of test generation on dynamically typed languages. Unlike strongly typed languages, generation in languages such as Python or Pharo faces the problem of missing type information. Similar to the previous tools, our work focuses on test generation using evolutionary search. However, we focus on Pharo, a dynamically typed language, and use a running example to collect type information used in the generation process.

⁵<https://www.evosuite.org>

⁶<https://randoop.github.io/randoop/>

5. Conclusion and Future Work

This paper presents SmallEvoTest, a tool to generate unit tests for any arbitrary Pharo class automatically. SmallEvoTest relies on a code example to extract argument type information and uses a just-in-time example collecting technique to combine method invocations and generate assertions. SmallEvoTest is a proposal for a foundation for automatic test generations, and our effort will be followed up with various points:

- *Conducting case studies*: Conducting case studies on representative classes of prominent Pharo systems is an obvious next step. This will help us illustrate some limitations of our approach and will help us identify actions to take to generate unit tests for large classes.
- *Improving assertions*: Many aspects of SmallEvoTest are based on immediate decisions taken from ad-hoc examples. In particular, the generation of assertions can be significantly improved. Incorporating tests about collections or structural similarities seems to be a reasonable step forward to produce.
- *Abstract template*: The objective of the generated tests is to cover a particular part of the code given a particular budget, expressed in terms of test methods and statements. As such, the generated tests differ from manually written tests. As a future work, we plan to incorporate an abstract template as a way to better structure the generated tests. Abstract templates are meant to generate tests that follow a particular structure, e.g., accessors must be invoked to properly initialize the object before invoking methods with business logic.

SmallEvoTest is available under the MIT License for the Pharo and GToolkit platforms.

Acknowledgments

Juan Pablo Sandoval Alcocer thanks ANID FONDECYT Iniciacion Folio 11220885 for supporting this article.

References

- [1] A. Bacchelli, P. Ciancarini, D. Rossi, On the effectiveness of manual and automatic unit test generation, in: 2008 The Third International Conference on Software Engineering Advances, 2008, pp. 252–257. doi:10.1109/ICSEA.2008.66.
- [2] G. Fraser, M. Staats, P. McMinn, A. Arcuri, F. Padberg, Does automated unit test generation really help software testers? a controlled empirical study, ACM Transactions on Software Engineering and Methodology (TOSEM) 24 (2015) 1–49.
- [3] J. S. Kracht, J. Z. Petrovic, K. R. Walcott-Justice, Empirically evaluating the quality of automatically generated and manually written test suites, in: 2014 14th International Conference on Quality Software, 2014, pp. 256–265. doi:10.1109/QSIC.2014.33.
- [4] J. M. Rojas, G. Fraser, A. Arcuri, Automated unit test generation during software development: A controlled experiment and think-aloud observations, in: Proceedings

- of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Association for Computing Machinery, New York, NY, USA, 2015, p. 338–349. URL: <https://doi.org/10.1145/2771783.2771801>. doi:10.1145/2771783.2771801.
- [5] G. Fraser, A. 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, ESEC/FSE '11, Association for Computing Machinery, New York, NY, USA, 2011, p. 416–419. URL: <https://doi.org/10.1145/2025113.2025179>. doi:10.1145/2025113.2025179.
 - [6] A. Panichella, J. Campos, G. Fraser, Evosuite at the sbst 2020 tool competition, in: Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops, ICSEW'20, Association for Computing Machinery, New York, NY, USA, 2020, p. 549–552. URL: <https://doi.org/10.1145/3387940.3392266>. doi:10.1145/3387940.3392266.
 - [7] G. Fraser, A. Arcuri, Whole test suite generation, IEEE Transactions on Software Engineering 39 (2012) 276–291.
 - [8] P. Tonella, Evolutionary testing of classes, in: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA '04, Association for Computing Machinery, New York, NY, USA, 2004, p. 119–128. URL: <https://doi.org/10.1145/1007512.1007528>. doi:10.1145/1007512.1007528.
 - [9] A. Sakti, G. Pesant, Y.-G. Guéhéneuc, Instance generator and problem representation to improve object oriented code coverage, Software Engineering, IEEE Transactions on 41 (2015) 294–313. doi:10.1109/TSE.2014.2363479.
 - [10] S. Panichella, A. Gambi, F. Zampetti, V. Riccio, Sbst tool competition 2021, in: 2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST), 2021, pp. 20–27. doi:10.1109/SBST52555.2021.00011.
 - [11] A. Gambi, G. Jahangirova, V. Riccio, F. Zampetti, Sbst tool competition 2022, in: 2022 IEEE/ACM 15th International Workshop on Search-Based Software Testing (SBST), 2022, pp. 25–32. doi:10.1145/3526072.3527538.
 - [12] G. Fraser, A. 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, ESEC/FSE '11, ACM, New York, NY, USA, 2011, pp. 416–419. URL: <http://doi.acm.org/10.1145/2025113.2025179>. doi:10.1145/2025113.2025179.
 - [13] C. Pacheco, S. K. Lahiri, M. D. Ernst, T. Ball, Feedback-directed random test generation, in: Proceedings of the 29th International Conference on Software Engineering, ICSE '07, IEEE Computer Society, USA, 2007, p. 75–84. URL: <https://doi.org/10.1109/ICSE.2007.37>. doi:10.1109/ICSE.2007.37.
 - [14] S. Vogl, S. Schweikl, G. Fraser, A. Arcuri, J. Campos, A. Panichella, Evosuite at the sbst 2021 tool competition, in: 2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST), 2021, pp. 28–29. doi:10.1109/SBST52555.2021.00012.
 - [15] A. Panichella, F. M. Kifetew, P. Tonella, Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets, IEEE Transactions on Software Engineering 44 (2018) 122–158. doi:10.1109/TSE.2017.2663435.
 - [16] J. Campos, Y. Ge, N. Albunian, G. Fraser, M. Eler, A. Arcuri, An empirical evaluation of evolutionary algorithms for unit test suite generation, Information and Software

Technology 104 (2018) 207–235. URL: <https://www.sciencedirect.com/science/article/pii/S0950584917304858>. doi:<https://doi.org/10.1016/j.infsof.2018.08.010>.

- [17] A. Panichella, S. Panichella, G. Fraser, A. Sawant, V. Hellendoorn, Test smells 20 years later: Detectability, validity, and reliability, *Empirical Software Engineering* 27 (2022). doi:10.1007/s10664-022-10207-5.
- [18] A. Panichella, F. M. Kifetew, P. Tonella, Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets, *IEEE Transactions on Software Engineering* 44 (2017) 122–158.
- [19] J. M. Rojas, M. Vivanti, A. Arcuri, G. Fraser, A detailed investigation of the effectiveness of whole test suite generation, *Empirical Software Engineering* 22 (2017). doi:10.1007/s10664-015-9424-2.
- [20] S. Lukasczyk, F. Kroiß, G. Fraser, Automated unit test generation for python, in: *Proceedings of the 12th Symposium on Search-based Software Engineering (SSBSE 2020, Bari, Italy, October 7–8)*, volume 12420 of *Lecture Notes in Computer Science*, Springer, 2020, pp. 9–24. doi:10.1007/978-3-030-59762-7_2.
- [21] S. Lukasczyk, F. Kroiß, G. Fraser, An empirical study of automated unit test generation for python, *CoRR abs/2111.05003* (2021). arXiv:2111.05003.