

TestEvoViz: Visual Introspection for Genetically-Based Test Coverage Evolution

Andreina Cota Vidaure
Semantics S.R.L.
Cochabamba, Bolivia
andycotvy@gmail.com

Evelyn Cusi Lopez
Semantics S.R.L.
Cochabamba, Bolivia
cusi.evelyn@gmail.com

Juan Pablo Sandoval Alcocer
Departamento de Ciencias Exactas e Ingenieria
Universidad Catolica Boliviana “San Pablo”
Cochabamba, Bolivia
jsandoval@ucb.edu.bo

Alexandre Bergel
ISCLab, Department of Computer Science (DCC)
University of Chile
Santiago, Chile
abergel@dcc.uchile.cl

Abstract—Genetic algorithms are an efficient mechanism to generate unit tests. Automatically generated unit tests are known to be an important asset to identify software defects and define oracles. However, configuring the test generation is a tedious activity for a practitioner due to the inherent difficulty to adequately tuning the generation process.

This paper presents TestEvoViz, a visual technique to introspect the generation of unit tests using genetic algorithms. TestEvoViz offers the practitioners a visual support to expose some of the decisions made by the test generation. A number of case studies are presented to illustrate the expressiveness of TestEvoViz to understand the effect of the algorithm configuration.

Index Terms—visualization, genetic algorithms, test generation

Artifact – <https://github.com/andreina-covi/ArtifactSSG>

I. INTRODUCTION

Automatic test generation is a crucial area in the field of software testing. It consists of generating executable unit test cases from a given source code base. A wide spectrum of techniques is commonly employed to generate tests, in particular fuzzing [1], test amplification [2], and genetic algorithms [3]. The unit tests generated are a valuable asset to identify the dead code or software defects as well as to define oracles [4].

This paper focuses on supporting the activity of test generation using genetic algorithms. EvoSuite¹ [3] is a popular genetically-based test generation tool. In this paper, we target the execution model proposed by EvoSuite, and as such, the scope of this paper is *offering a visual support tool to (i) understand the final result of a test generation and (ii) comprehend the process of getting to this result.*

The effort related to EvoSuite has significantly strengthened the field of genetically-based test generation. EvoSuite is considered a reference in the field and it has remarkable traction by using genetic algorithms to generate tests. However, it is surprising to see that EvoSuite does not provide

much tooling for understanding and assessing how tests are effectively generated. In particular, EvoSuite does not provide any mechanism to precisely expose the decision made by the genetic algorithm. As a consequence, understanding and characterizing the genetic algorithm execution is difficult. Such a situation may contribute to an inadequate tuning of the unit test generation algorithm. Adequately configuring a genetically-based test generation algorithm is difficult, in particular, (i) determining whether some *hyperparameters are properly chosen* or (ii) adequately identifying the algorithm termination condition to end the test generation algorithm are two notoriously challenges that practitioners must face to enjoy generated unit tests of a good quality.

We hypothesize that the difficulties in configuring a genetically-based test generation stem from the lack of *introspection mechanism* related to the algorithm execution.

TestEvoViz. We propose TestEvoViz, a visual introspection mechanism for genetically-based test generation. TestEvoViz visually represents the execution of the test generation, with the objective of assisting a practitioner understand decisions made by the genetic algorithm.

Figure 1 gives an example of TestEvoViz on a generation of unit test for the classical class `Stack`, describing a stack data structure. The visualization reads from *top* to *bottom* in which each line represents an iteration of the algorithm. TestEvoViz provides a range of glyphs detailing some aspects of the test generation. The figure shows that the test evolution goes through 6 iterations.

TestEvoViz is composed of three panels. The left-most panel indicates the contributions made for each of the 6 iterations. The contribution of each iteration is expressed using a spark circle [5], which summarizes three metrics related to test coverage: a big spark circle indicates a significant contribution of the generation in terms of covered code. The panel located in the middle represents the evolving unit tests that contribute to the final iteration. The right-most panel plots the evolution

¹<http://www.evosuite.org>

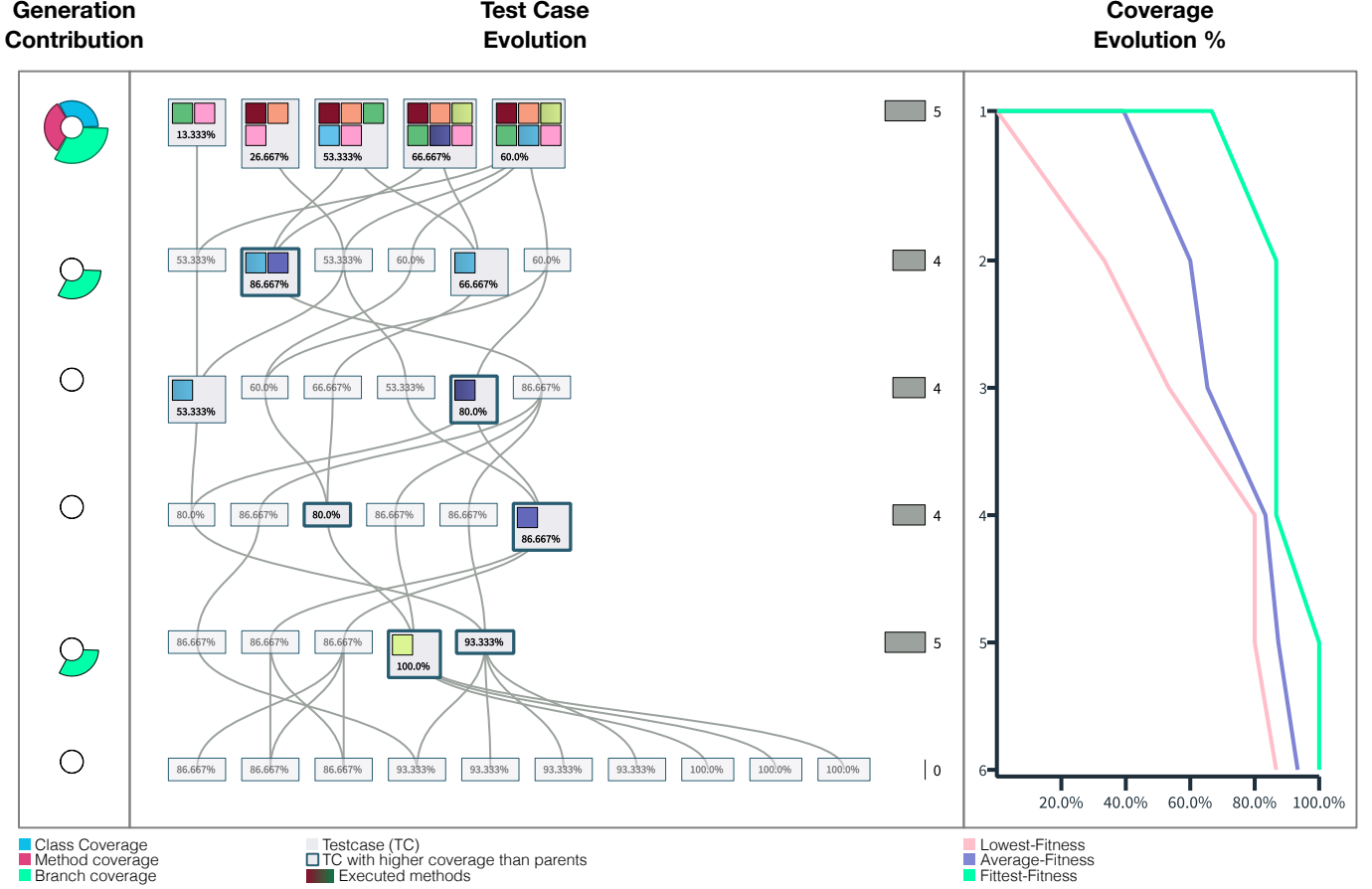


Fig. 1. TestEvoViz - Illustrating example: Test generation process for the `Stack` class. Left panel shows the coverage variation at project level between a given generation and its predecessor. Each box inside the middle panel represents a generated test. Links associate each test with their parents. A strong box border highlights tests that have greater coverage than their parents. The value of each box gives the percentage of code covered by the generated test. Inner boxes depict methods from which their test coverage changes with regard to their parents. Right panel reports the coverage evolution along generations by rendering the average, lowest and fittest coverage reached in each generation.

of test coverage evolution in terms of the best, average, and worse fitness. These curves are relevant to assess the diversity of the genetic information in the unit tests at each iteration of the algorithm. This right-most panel indicates that the generated tests covers 100% of the base component under test.

We have applied TestEvoViz to a number of non-trivial examples. TestEvoViz helps us characterize the behavior of the genetic algorithm.

Outline. The paper is structured as follows: Section II gives the necessary background to readers unfamiliar with genetically-based test generation; Section III describes the TestEvoViz visualization and the introspection mechanism; Section IV presents some examples that illustrates TestEvoViz in practice; Section V presents some real world case studies that highlight the benefits of TestEvoViz; Section VII gives an overview of the works related to this paper; Section VIII concludes and presents our future work.

II. BACKGROUND: GENETICALLY-BASED UNIT-TEST GENERATION

A. Unit-Test Generation

A number of techniques have been proposed to automatically generate tests [3], [4], [6]. In this paper, we voluntary focus on *EvoSuite* [3], a testing tool suite, which uses a genetic algorithm to generate unit tests. Unit tests are evolved by applying genetic operation to maximize the test coverage of a class belonging to the base application code. Such a class represents the target component *EvoSuite* is generating tests for. The coverage of the target class is considered the fitness function that the genetic algorithm is optimizing. A population of tests is evolved by *EvoSuite* using primitive genetic operations.

Each individual of the population is a test, which is composed of a number of executable source code statements. The statements contained in each test represent the genetic information, commonly referred to as chromosome. There are four kinds of statements considered by *EvoSuite*: *primitive*,

which represents a literal value (e.g., number, boolean, string), *constructor* to create an object from a class of the application under test, *method call* to send a message to an object, and *access field* to access an object variable. After having built the tests, another algorithm generates assertions by using values produced by the statements.

Each test contained in an unit test is composed of an initialization code portion and a set of assertions. Figure 2 gives an example of a test method. Test methods are generated to maximize the execution coverage and the whole test generation is oriented to executing the largest portion of the target class. As such, TestEvoViz does not represent assertions.

Generated Unit Test	statement kind
int var0 = 0;	primitive
int var1 = 1;	primitive
Point var2 = new Point(var0,var0);	constructor
Point var3 = new Point(var1,var0);	constructor
double var4 = var2.distance(var3);	method call
int var5 = var2.x;	access field
assertEquals(var5,var0);	Assertion
assertEquals(var4,1);	Assertion
assertEquals(var2.toString(),"0,0");	Assertion

Fig. 2. Unit test as individual of the Population

Initial Population. First, the algorithm creates N tests, and each test has M randomly generated statements. Each statement tries to benefit from the previous statements contained in the same test by using variables previously defined. Figure 2 gives an example of a test in which the third statement uses the variable `var0` defined in the first statement.

Evolution. Once the initial population is defined, four steps are performed to produce a new iteration, and therefore a new population of evolved tests, by the algorithm:

- *Coverage measurement* – Each test is executed and the code coverage of that test is measured through three different metrics, as we will see later on.
- *Selection* – In a given population of tests, only the better-performing tests are evolved. The selection algorithm determines which tests have to be evolved. Many algorithms are available (e.g., ranking selection, roulette, tournament).
- *Crossover* – The genetic information of two selected unit tests are combined using the crossover genetic operation. A crossover between two tests consists in merging their statements to generate two new tests.
- *Mutation* – The tests resulting from a crossover may be randomly altered using a mutation genetic operation. A mutation replaces a statement with a new one or a variation of it. Numerous mutation operators can be applied, including changing a parameter for another (e.g.,

replacing a variable name for another or changing a primitive literal value for another). Mutations are necessary to produce diversity in the genetic information.

These operations are performed multiple times to produce a new and evolved generation of unit tests.

B. Challenges

The complexity of the underlying genetic algorithm makes the activity of generating test difficult and tedious for a practitioner. In particular, a number of technical issues have to be considered in order to properly generate unit tests of a good quality:

- *Hyperparameter tuning* – A hyperparameter is a parameter whose value is used to control the test generation process. Numerous hyperparameters are associated with genetically-based test generation: statement mutation rate, size of the population, selection algorithm, crossover rate, just to name a few. Identifying adequate hyperparameter values is a process that typically follows a try-and-adjust fashion.
- *Stopping the genetic algorithm* – Generating unit tests may take hours or even days for a non-trivial software component. A central question is: When to stop the evolution of the unit tests? This question is hard to answer in practice. The behavior that is commonly followed by practitioners is to maximize the number of generations in order to reach the best result. However, it frequently happens that most of the best-performing tests (i.e., the ones with the a high coverage) are generated in an early iteration. Furthermore, unit test generation is a computationally intensive process and avoiding unnecessary iterations has a significant practical impact.

These two problems cannot be easily solved. The coming section presents TestEvoViz, which alleviates these problems by providing to practitioners essential information about the test generation algorithm execution.

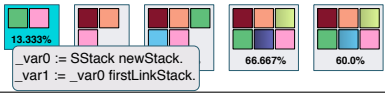
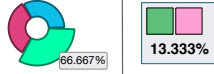


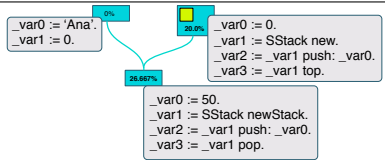
III. TESTEVOVIZ

We propose *TestEvoViz*, a visual approach to represent the generation of unit tests using genetic algorithms. TestEvoViz visually introspects the algorithm internal to let a practitioner better understand decisions taken by the algorithm. TestEvoViz has three main visual components to convey different aspects regarding the iterative evolution of the population of unit tests. This section describes a data model and each one of these components using as example the Figure 1, which illustrates the test generation for the `Stack` class. Table I details the relation between the genetic algorithm concepts and the proposed visualization.

A. Data Model and Introspection

Our approach is designed to visualize how test cases are evolving across generations in order to achieve a higher coverage. Let $G_n = \{g_0, g_1, \dots, g_n\}$ be the set of populations created by the genetic algorithm, where g refers to a population

TABLE I
MAPPING GENETIC ALGORITHM CONCEPT IN TESTEVOVIZ

Initialization	Population is composed by N tests (chromosome), which are composed by M statements (genes).	
Fitness	The fitness is given by the branch coverage of each test and it is shown at the bottom of each node. In addition, the left panel shows the class, method and branch coverage variation of each generation. Each of these metrics is associated to a ring in a spark circle.	
Selection	Each node in the middle panel represents a test that contributes to the final generation. These tests were selected during the generation process using a selection criteria (i.e., rank selection). Our visualization also shows the number of nodes that were discarded in each generation.	
Crossover	Tests that participate in a crossover operation are visually linked to their child.	
Mutation	Statements that were mutated after a crossover operation may be detected by contrasting the source code of a given test with the source code of their parents.	

of tests: the numerical subscript is the iteration index, and n is the number of iterations. The initial random population is denoted g_0 . Each population g_k consists of m tests $g_k = \{t_0, t_1, \dots, t_m\}$, where m is the size of the population. A tuple (t_i, g_j) defines a test i of the population in the iteration j . Let $ancestors(t_i, g_j)$ be the set of ancestors of the tuple (t_i, g_j) , each tuple (t_i, g_j) may have one or two ancestors. We define $ancestors(t_i, g_j)$ as the tests of the previous population in iteration $j - 1$ that participate in the creation of the test t_i .

We have augmented the genetic algorithm to emit events at relevant steps during its execution, e.g. before and after each iteration, application of a genetic operation. These events are used to build a detailed logging facility from which TestEvoViz extracts relevant information to build the visualization.

B. Generation Contribution

The left-hand side panel of TestEvoViz contains a *spark circle* for each generation of the evolution (Figure 3, left-hand side). A spark circle is a small bar chart which is drawn in a circular fashion. Our approach uses a spark circle with three ring sections. Each spark circle summarizes the coverage variation of a given population g_j compared from its previous population g_{j-1} at three levels of granularity:

- **Branch Coverage** – Let $Bcov(g_j)$ be the ratio between the number of executed branches by all the tests of the generation and the number of existing branches in the system. The total number of branches is the sum of the branches of all methods in the application under test.

- **Method Coverage** – Let $Mcov(g_j)$ be the ratio between the number of executed methods and the number of methods of the application under test.
- **Class Coverage** – Let $Ccov(g_j)$ be the ratio between the number of classes that have at least one method executed regarding all the classes in the application under test.

We define the *coverage variation* between g_j and g_{j-1} as follows:

$$\Delta cov(g_j, g_{j-1}) = (cov(g_j) - cov(g_{j-1})) / (cov(g_{j-1}))$$

This definition is used to measure coverage variation at branch ($Bcov$), method ($Mcov$) and class ($Ccov$) level. The execution of a generated test case may cover different methods and classes of a system. Therefore, height of each ring section is associated to the variation of the three coverage metrics: branch coverage variation (green section), method coverage (red section), and class coverage variation (blue section), as indicated in Figure 3. In Figure 1, we see that the evolution brought by the test in generations 1, 2, and 5, contribute to significant increment the branch coverage. In generation 1 we also see that the method coverage and the class coverage reached its maximum since these two metrics did not change in the later iterations.

C. Test Case Evolution

The middle panel of TestEvoViz (Figure 1) details the unit test evolution along the iterations.

Nodes. Each node represents a test case of a particular generation (t_i, g_j) . Tests at a given iteration are horizon-

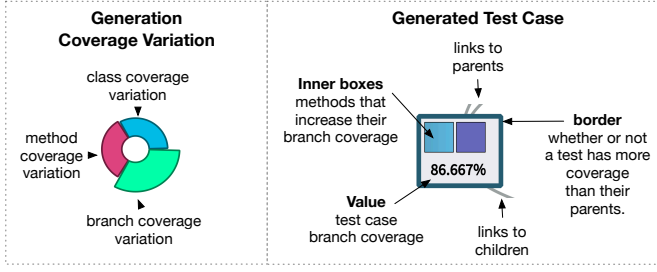


Fig. 3. Spark circle and node glyph

tally aligned as represented in Figure 1 and Figure 4. Let $Bcov(t_i, targetClass)$ be the ratio between the number of executed branches in the target class regarding the total, and $Bcov(t_i, m)$ the branch coverage of a method m . We define the visual cues associated to a unit test node (Figure 3) as follows:

- **Border** – A thick border indicates that a test case (t_i, g_j) has a higher branch coverage than its ancestors $Bcov(t_i, targetClass) > Bcov(t_h, targetClass)$, for all $t_h \in ancestor(t_i, g_j)$. If the coverage remains the same or does not improve then the box has a thin border.
- **Inner boxes** – Each colored inner box represents a method m of the target class that improves its branch coverage regarding the unit test ancestors $Bcov(t_i, m) > Bcov(t_h, m)$, for all $t_h \in ancestor(t_i, g_j)$. To differentiate the methods, each method of the target class has a unique color. Note that different tests may increase their coverage of the same methods.
- **Value** – The bottom value gives the class branch coverage obtained after executing a given test case $Bcov(t_i, targetClass)$.

Edges. Edges connect tests and indicate the historical evolution of these tests. An edge joins a unit test to its ancestors. A unit test may have one or two ancestors. A unit test with two ancestors means that the unit test is the result of a crossover operation of two previous unit tests. In some cases, a node has only had one ancestor, because either (i) the unit test was the best of the generation and it survives due to the elitism strategy; (ii) or produced children have a lower coverage than their parents, in this case, the algorithm chooses to let one of the two parents survive in the next generation.

Killed unit tests. To not overload the visualization, TestEvoViz does not represent unit tests that do not contribute to the final generation. During the evolution, many generated unit tests are poorly performing (i.e., have a low coverage), and therefore are killed (i.e., not considered for selected for being combined with other unit test). The amount of killed unit tests for each generation is represented as a horizontal bar, located on the right hand side of the middle panel (Figure 1).

Interaction. TestEvoViz provides a number of interactions to inspect the source code and track a test case genealogical tree. Clicking on a node highlights their ancestors. Hovering the

mouse over a unit test shows the generated test code, and hovering the mouse cursor over an inner box shows the source code of the corresponding method. Figure 4 shows all the ancestors of a test, and also shows the source code of the selected test.

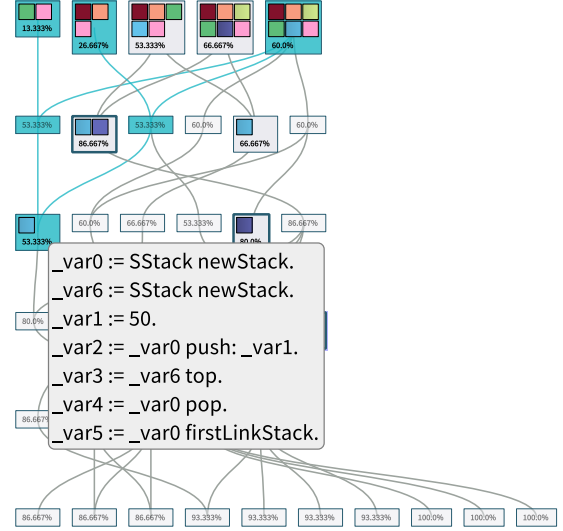


Fig. 4. Highlighting ancestors and obtaining source code

D. Coverage Evolution

The panel located on the right-hand side of TestEvoViz indicates the evolution of the coverage by means of three curves (Figure 1): the fittest unit test per generation (green line), the average of the unit test coverage in a generation (blue line), and the worse unit test per generation (pink line). The distance between the worst and the fittest helps in assessing the overall health of the generation: more distance between these two curves, more diverse in the genetic information and the population are.

IV. EXAMPLES

This section describes an application of TestEvoViz to introspect the test generation of two classes of the *Pharo* programming language: *Stack* and *DataFrame*. These are two popular data structures implemented in *Pharo*. While the first one is a classical linear data structure, the second one is a two-dimensional structure commonly used for data analysis. We use TestEvoViz to generate tests for these classes and introspect the generation process. Figure 1 and Figure 5 depict the results obtained for *Stack* and *DataFrame*, respectively. The following paragraphs detail the test generation as executed by EvoSuite.

Improving Ancestors Coverage. Along the evolution there are a number of nodes that have a better coverage than their ancestors. These tests are the ones that have a thick border. The tests for *Stack* (Figure 1) show that eight nodes (i.e., tests) perform better than their parents (i.e., the children tests have a higher coverage than their parents). Each iteration generates

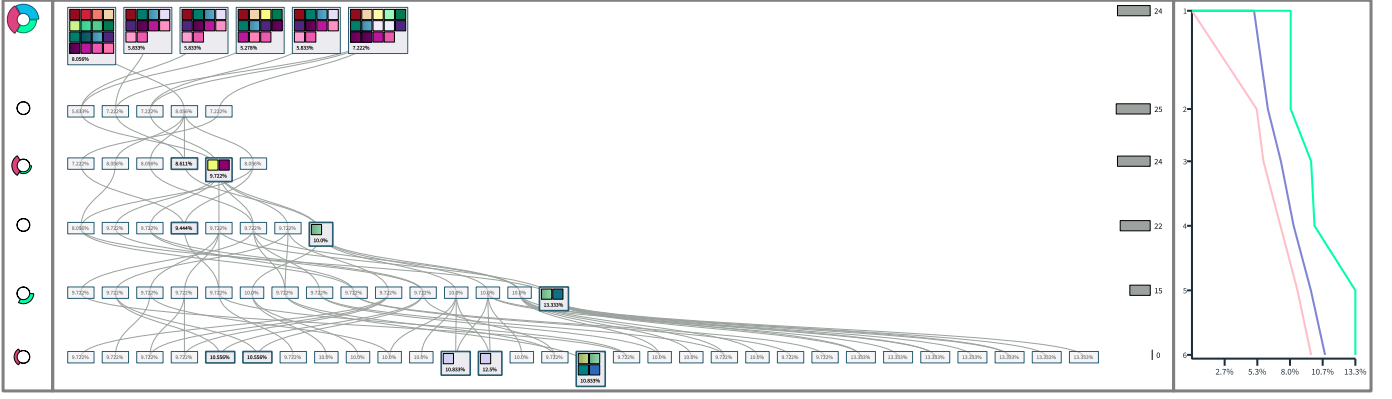


Fig. 5. TestEvoViz on the DataFrame class

exactly two new test cases with a better coverage. On the other hand, the evolution of tests for `DataFrame` (Figure 5) has also ten nodes, however the difference is that five of these ten nodes were discovered in the last iteration. We can categorize these nodes in two:

- *With inner boxes* – Nodes with inner boxes represent test cases that cover new branches or methods regarding their ancestors. The color of inner boxes helps us to differentiate this situation. If a color does not appear before, then it indicates that a new method is discovered, otherwise, a new branch of a previously executed method is discovered.
- *Without inner boxes* – A node without any inner box represents a test case that has a better coverage than its parents, but did not cover any new method or branch. This happens when its parents cover different branches of the target class, and their child covers part of all these branches together due the crossover mechanism.

Improving Generation Coverage. Although some tests of an iteration have a better coverage than their parents, they may discover new branches that may be already covered by the others tests in the same iteration. As such, the algorithm is discovering the same findings multiple times. The generation contribution panel helps us identify this situation. Figure 1 shows this situation in generation 3 and 4 although there are tests that cover new branches regarding their parents. The coverage of the population does not increase at all. Therefore, these tests cover branches already covered by other individuals of the population. On the other hand, in the second and fifth iteration the new tests discover new branches (i.e., not previously discovered). This fact is also reflected in the coverage evolution component, every time that a new test covers new branches, both the fittest and average coverage of the population increase.

Discovering Dependencies. Sometimes, discovering a new branch is due to code statements that involve method calls to method or classes that were not covered in the previous iterations. This fact is also reflected in the generation contribution panel, which shows the coverage variation at method

and class level. For instance, Figure 1 shows that all but the last iterations discover new branches that did not involve any new method or class. However, Figure 5 shows that the method coverage improves in the iterations 3 and 6, meaning that new methods have been called by a given generation of tests.

Discarding weak tests. In each generation, the selection algorithm discards tests that do not participate in the creation of the new population. This fact is shown by the gray bars positioned at the right side evolution component. Since, the purpose of the selection algorithm is to discard weak tests from the population (i.e., poorly performing with a low coverage). The selection algorithm is related to the metric lowest coverage on the population, which is shown by the coverage evolution component. For instance, Figure 1 and Figure 5 show that the selection algorithm does a good job, because at every generation, tests with a low coverage are excluded, and the lowest coverage is increasing. A particular situation is shown in the second iteration in Figure 5, because, none of the tests of that iteration improve their coverage. However, the lowest coverage increases. This means that even though there were no improvement the algorithm discards test cases with low coverage.

V. CASE STUDIES

This section presents two case studies on which we analyze the generation process of two *Pharo* projects to address two questions:

- *Q1: What are the effects of the number of statements on the test generation process?*
- *Q2: What are the effects of the population size on the test generation process?*

A. Regex

Regex is a standard *Pharo* library to parse and match regular expression expressions. In this case study, we use the class *RxMatcher* as a target class. *RxMatcher* is a recursive regular expression matcher that has 27 methods.

Baseline. For this case study, we use a base configuration as follows: *number_of_statements* = 5,

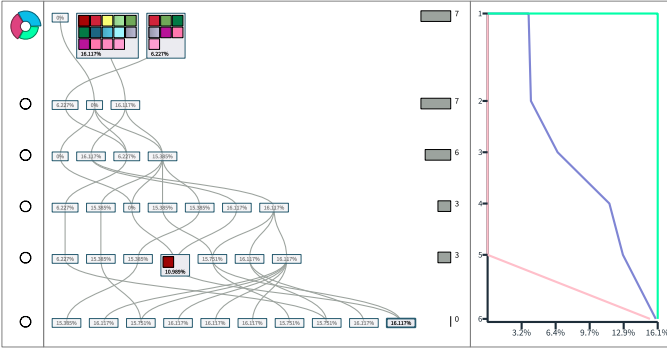


Fig. 6. TestEvoViz – Regex project; *number_of_statements* = 5; *number_of_iterations* = 5; *selection_algorithm* = *rank_selection*; and *population_size* = 10

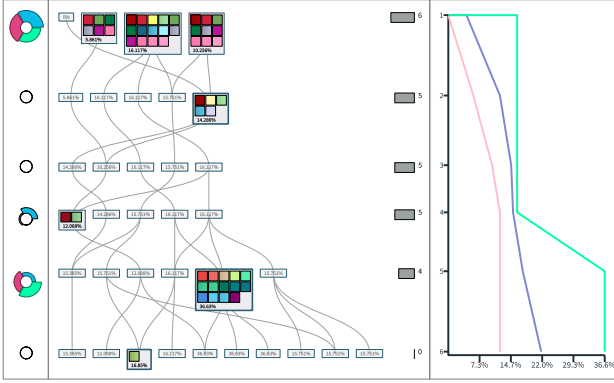


Fig. 7. TestEvoViz – Regex project; *number_of_statements* = 3; *number_of_iterations* = 5; *selection_algorithm* = *rank_selection*; and *population_size* = 10

number_of_iterations = 5, *selection_algorithm* = *rank_selection*, and *population_size* = 10. Figure 6 gives the results of running the algorithm with previous configuration. As we can see, most of the methods and branches are covered at the beginning of the first iteration. Only in the fifth iteration a test case improves its coverage regarding their parents, but that branch was already covered since the spark circle of that iteration does not present any coverage variation. After five iterations the tests with the highest branch coverage is 16%. In this particular case study, there are generated tests with 0% of coverage. This is because the algorithm only creates object creation statements basically calling to class default constructors.

Q1: Number of Statements. Figure 7 depicts the generation process using the same base configuration, with the exception that this time we set the number of statements as three. Figure 7 shows that more tests of the first iteration survive regarding the baseline. However, the first iteration test cases have similar coverage than in the baseline. This means that the first iteration, although having less number of lines, covers the same amount of branches.

Along the evolution four tests have more coverage than their ancestors, one may notice this by searching for nodes

with a thick border. The generation contribution panel shows that the fourth and fifth iteration discover new methods and branches not previously discover. This particular visualization shows that the crossover operations between individual with less statements achieve a higher coverage compared to the baseline. With this configuration, the best generated test case covers 36% branches of the target class, which is more than baseline.

Q2: Population Size. Figure 8 depicts the generation process using the same base configuration, with the exception that this time we use a population size of 20. Figure 8 shows that in the crossover operation between two test results in a new test case that covers new branches, methods and classes. This fact is indicated by the spark circle of the third iteration. Similarly to the baseline, the visualization shows that there are few tests that have better coverage than their ancestors. But in this case, the algorithm found a new test case which got a better coverage than the baseline.

B. NeoJSON

NeoJSON is the standard JSON reader and writer of the Pharo programming language. In this case study, we generate tests for the class *NeoJSONObjectMapping* which has 17 methods.

Baseline. We use the following base configuration: *number_of_statements* = 20; *number_of_iterations* = 10; *selection_algorithm* = *rank_selection*; and *population_size* = 20. Using a greater number of iterations and statements has the effect to produce a larger visualization. Figure 9 shows the test evolution process for the class *NeoJSONObjectMapping*. As we can see, most of the target class is covered with the tests from the first iteration. Iterations 3 and 5 slightly increment the branch coverage. This fact is showed by the coverage evolution and generation contribution panel. There were four generations (between six and ten) where the algorithm could not evolve until the last iteration. Though the last generation has a better coverage than their parents, it does not discover any new branches.

Q1: Number of Statements. To address the first question we increase the *number_of_statements* from 10 to 20. Figure 10 shows the visualization result of this change. First, we notice that (i) the last generation has a greater coverage, and (ii) most of the branches are discovered by the first three generations. Regarding the baseline that most of the coverage is discovered in the first five iterations. Therefore, we conclude that in this particular case a greater number of number of statements helps discover more branches quickly. In addition the last generation covers more portion of the target class than the baseline.

Q2: Population Size. To address the second question, we use the base configuration but use a population size of 30 instead of 20. Figure 11 shows the resulting evolution process. Since, the population is larger, there are more tests that survive on the first generation. The visualization shows that the coverage is

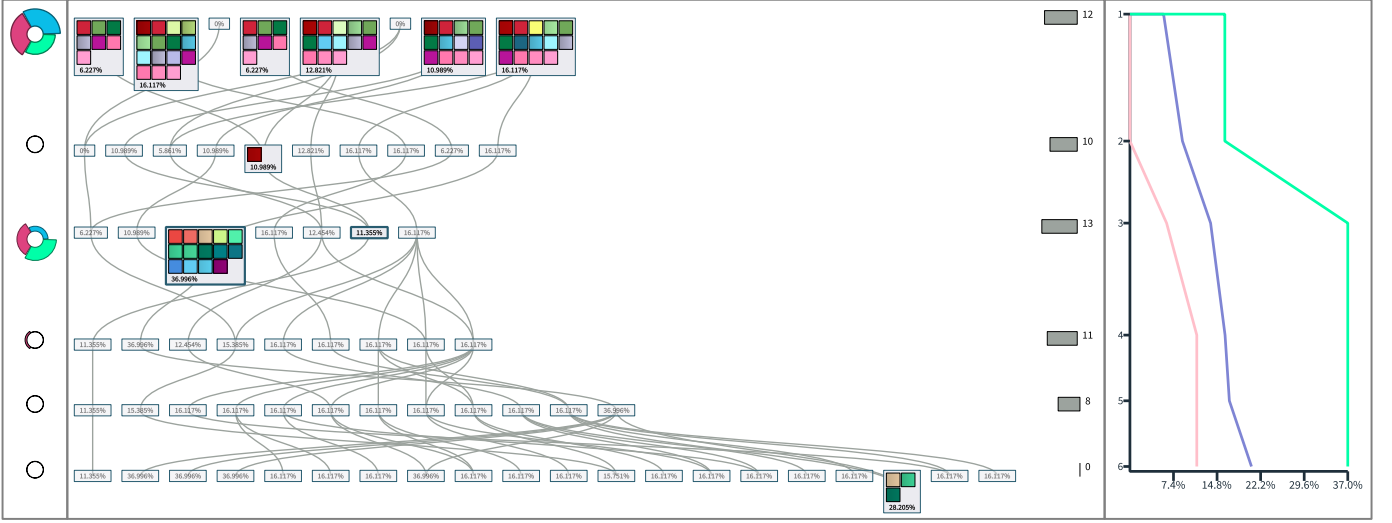


Fig. 8. TestEvoViz – Regex project; *number_of_statements* = 5; *number_of_iterations* = 5; *selection_algorithm* = *rank_selection*; and *population_size* = 20

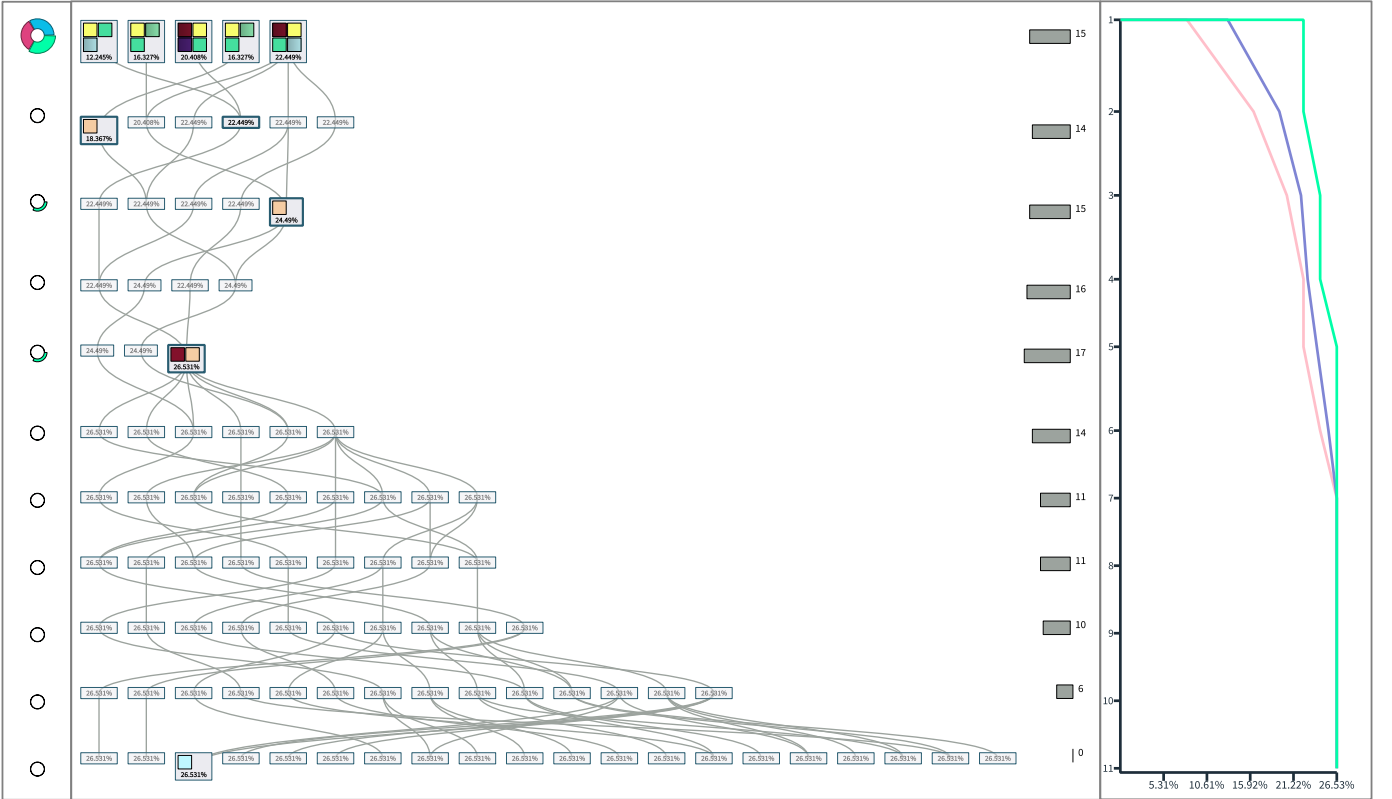


Fig. 9. TestEvoViz – NeoJSON project; *number_of_statements* = 10; *number_of_iterations* = 10; *selection_algorithm* = *rank_selection*; and *population_size* = 20

improved by four iterations (second, third, fourth and seventh), which is two more than the baseline. There are also more tests that have a better coverage than their parents. However, the coverage reached by the last generation is 12 % bigger than the baseline.

VI. DISCUSSION

Scalability. TestEvoViz uses a grid layout, which makes the overall visualization size depend on the population size and the number of iterations. Therefore, a larger visualization typically requires scrollbars which may involve more interaction from a practitioner to enjoy the visualization. To mitigate the negative

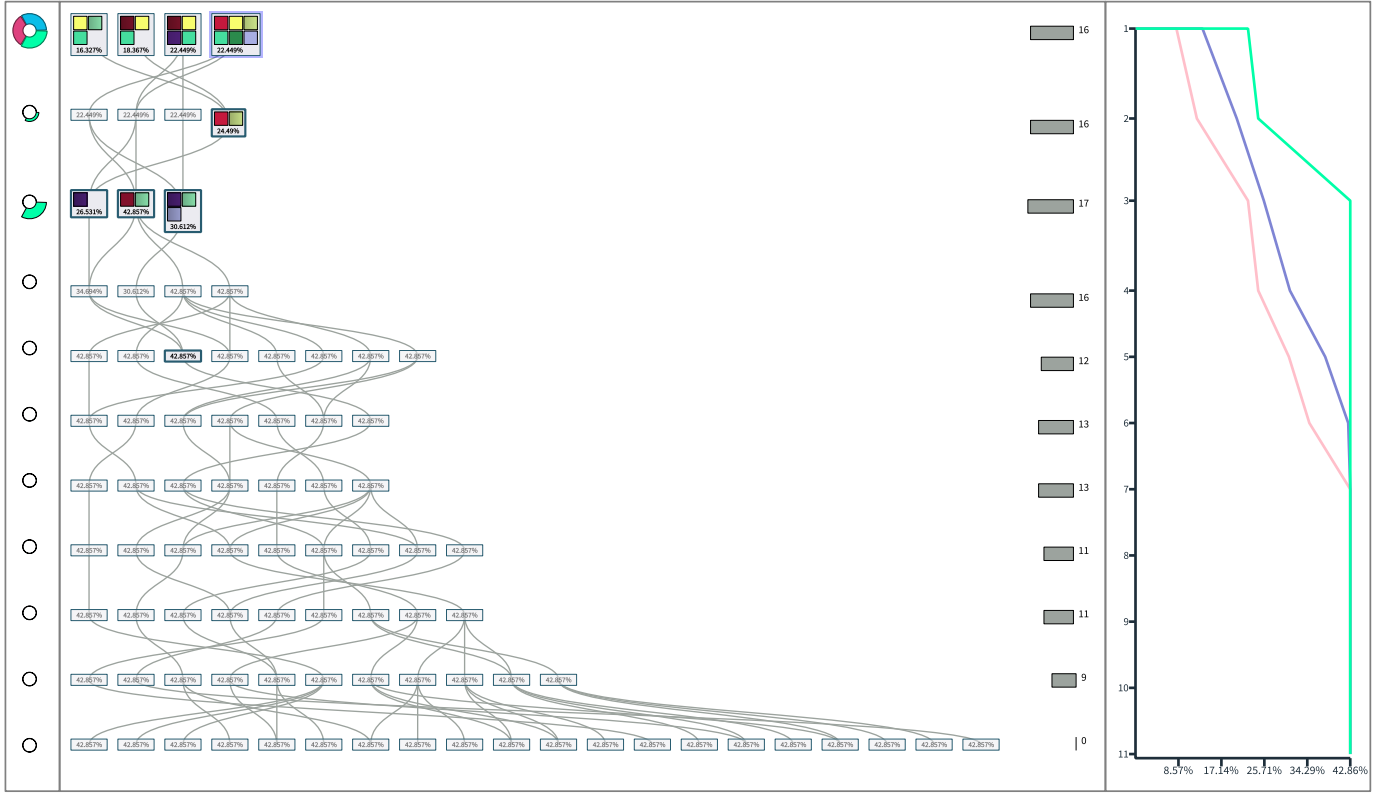


Fig. 10. TestEvoViz – NeoJSON project; *number_of_statements* = 20; *number_of_iterations* = 10; *selection_algorithm* = *rank_selection*; and *population_size* = 20

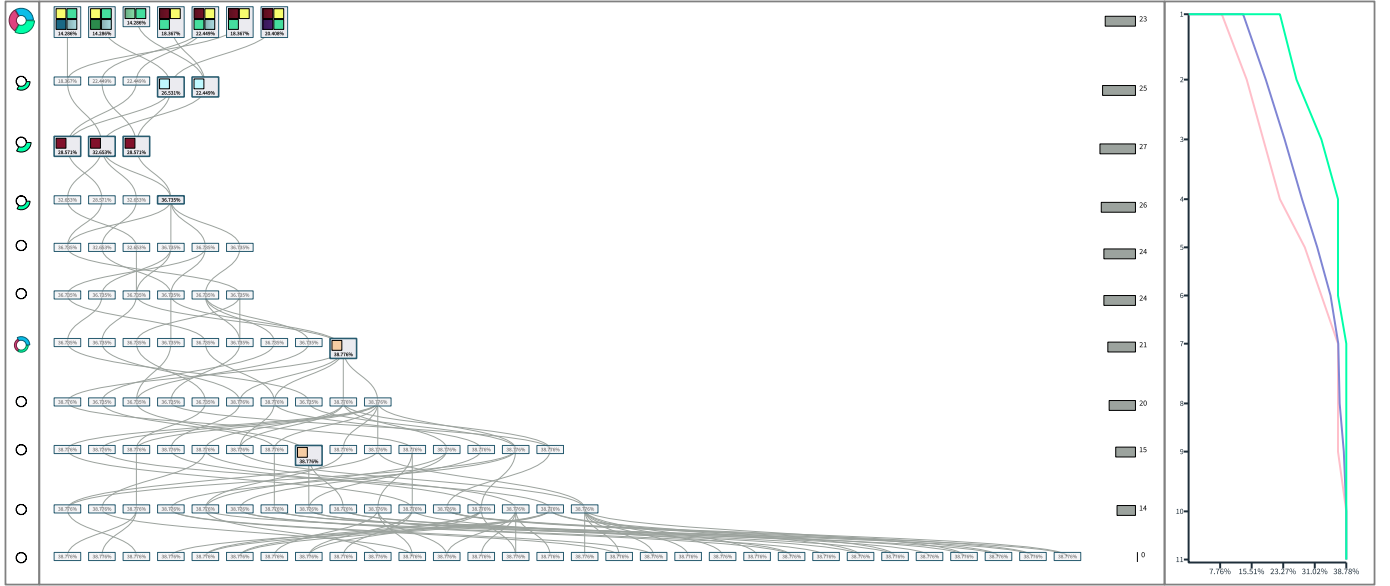


Fig. 11. TestEvoViz – NeoJSON project; *number_of_statements* = 10; *number_of_iterations* = 10; *selection_algorithm* = *rank_selection*; and *population_size* = 30

effect of this situation, our tool offers zoom-in and zoom-out facilities using the mouse wheel. We argue that even though the size of the nodes may be small when zoomed out, patterns remain identifiable.

Method Colors. We assign a particular color to each method of the target class. This color helps identify whether methods are discovered multiple times by the algorithm or whether the test covers new branches in method. In presence of a large number of methods, such an approach could lead to reduced

visualization readability. In this case, hovering the mouse gives a contextual popup window information to precisely identify a method.

Pharo Implementation of EvoSuite. Our visualization is implemented over a test generator for Pharo called *SmallSuiteGenerator*². The main difference between our implementation and *EvoSuite* is about resolving type information to drive the test generation. *EvoSuite* operates in Java, which is statically typed (i.e., each variable has a static type). Since Pharo is a dynamically typed language (like Python and JavaScript), *SmallSuiteGenerator* has to use various strategies and heuristics to extract type information from executing a Pharo application. Currently, *TestEvoViz* is not representing collected or inferred type information that application uses to generate tests.

Generalization. Our visualization helps developers introspect the generation process to understand how the algorithm is performing. As we see in our case studies, a simple variation in the parameters may significantly impact the algorithm behavior. However, it is important to clarify that the behavior also depends on many other variables, for instance, the target class and the complexity of their methods. Therefore, it is not possible to generalize the findings outside the configuration on which the algorithm was run.

VII. RELATED WORK

Genetic algorithms have been proposed in the 60s, since then, numerous efforts have been made to improve and evaluate genetic algorithms. Most of the existing works use standard visualizations (i.e. line charts and box plots) to show the evolution of a number of metrics along evolution to describe each generation. The spread of the fitness along each individual of a generation is usually represented using chart as we do in the third panel of *TestEvoViz*. A number of detailed visualizations have been proposed to better understand the evolution process.

Our visualization was inspired by a number of visual techniques even though they have a different purpose. We combined and adapted these to build our proposed approach. We employ spark circle [5] to highlight coverage variations between iterations. We use a Cartesian layout to relate generated tests with their corresponding iterations [7]–[10]. We associated a number of metrics to each node inspired in polymetric view [11]–[13] and edge lines were inspired from hierarchical bundle edges [14].

Representing generated tests with target class’ executed methods was tailored of [15], which uses boxes to represent methods. In other hand, relationships between nodes with their ancestors are represented as edges [15], [16]. *TestEvoViz* also shares similarities with [16] to represent fitness progress through iterations.

Hart *et al.* [16] propose an ancestry view, to render all the ancestors of the best individual after the generation process,

using a tree layout and coloring nodes based on a number of individual properties (i.e gene values, fitness, and gene origins). Romero *et al.* [17] use color maps to visualize the individuals and chromosomes of the population.

Farooq *et al.* [18], [19] propose a visualization for interactive genetic algorithms (IGA), IGA combines the evolution mechanism with user’s intelligent evaluation, where users help the algorithm in the evolution process. In particular, this visualization helps users to decide the generation for interaction. It uses a two axis dot plot visualization, where the horizontal axes are the generation number, and the vertical axes the coverage of each individual all generations.

Ito *et al.* [20] proposed the use of pseudo-color to visualize binary-code individuals of the population using pseudo-color assigning a red pixel to chromosomes that represent “1”, and a blue pixel to “0”.

Tomida *et al.* [21] proposes a technique to visualize the evolution process of automated program repair. It is based in a tree layout showing the code genealogy. It highlights the nodes according to the operations and variants performed in individuals of the population. These operations are particular to tasks of automated program repair. This work is related to our effort.

At the difference of these works, our approach focuses on genetically-based test coverage evolution. Therefore, our visualization renders information highly related to test evolution, their operations and properties. As far as we know, this is the first approach to help developers understand the test generation process along the genetic algorithm.

VIII. CONCLUSION AND FUTURE WORK

TestEvoViz introspects a test generation algorithm execution and uses a visualization to expose some aspects of the generation process. The resulting visualization may be exploited by practitioners to adjust the algorithm configuration.

We also present some situations in which *TestEvoViz* is able to support a non-trivial analysis of the test generation.

As future work, we will support differences of algorithm execution. Currently, *TestEvoViz* visualizes the execution based on one single algorithm configuration. In the future, we will make *TestEvoViz* show differences between multiple executions of the algorithm.

ACKNOWLEDGMENT

We are deeply grateful to Lam Research (4800054170 and 4800043946). Alexandre Bergel is grateful to the FONDECYT Regular project 1200067 for having partially sponsored the work presented in this article. We thank Renato Cerro for his help reviewing an early draft of the manuscript.

REFERENCES

- [1] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, “The fuzzing book,” in *The Fuzzing Book*. Saarland University, 2019, retrieved 2019-09-09 16:42:54+02:00. [Online]. Available: <https://www.fuzzingbook.org/>

²<https://github.com/OBJECTSEMANTICS/SmallSuiteGenerator>

- [2] B. Danglot, O. Vera-Perez, Z. Yu, A. Zaidman, M. Monperrus, and B. Baudry, "A snowballing literature study on test amplification," *Journal of Systems and Software*, vol. 157, p. 110398, 2019. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0164121219301736>
- [3] G. Fraser and A. Arcuri, "Evolutionary generation of whole test suites," in *International Conference On Quality Software (QSIC)*. IEEE Computer Society, 2011, pp. 31–40.
- [4] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," in *Proceedings of the ACM International Symposium on Software Testing and Analysis*, ser. ISSTA '10. ACM, 2010, pp. 147–158. [Online]. Available: <http://doi.acm.org/10.1145/1831708.1831728>
- [5] J. P. Sandoval Alcocer, H. Camacho Jaimes, D. Costa, A. Bergel, and F. Beck, "Enhancing commit graphs with visual runtime clues," in *2019 Working Conference on Software Visualization (VISOFT)*, 2019, pp. 28–32.
- [6] A. Arcuri and G. Fraser, "On parameter tuning in search based software engineering," in *Proceedings of the Third International Conference on Search Based Software Engineering*, ser. SSBSE'11. Springer-Verlag, 2011, pp. 33–47. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2042243.2042252>
- [7] M. Lanza, "The evolution matrix: Recovering software evolution using software visualization techniques," *International Workshop on Principles of Software Evolution (IWPSE)*, 09 2001.
- [8] F. Beck, M. Burch, C. Vehlow, S. Diehl, and D. Weiskopf, "Rapid serial visual presentation in dynamic graph visualization," 09 2012, pp. 185–192.
- [9] J. P. Sandoval Alcocer, A. Bergel, S. Ducasse, and M. Denker, "Performance Evolution Blueprint: Understanding the impact of software evolution on performance," in *Proceedings of the 1st IEEE Working Conference on Software Visualization*, ser. VISSOFT. IEEE, 2013, pp. 1–9.
- [10] J. P. Sandoval Alcocer, F. Beck, and A. Bergel, "Performance Evolution Matrix: Visualizing performance variations along software versions," in *Proceedings of the 7th IEEE Working Conference on Software Visualization*, ser. VISSOFT, 2019.
- [11] M. Lanza and S. Ducasse, "Polymetric views - a lightweight visual approach to reverse engineering," *IEEE Transactions on Software Engineering*, vol. 29, no. 9, pp. 782–795, 2003.
- [12] A. Bergel, F. Bañados, R. Robbes, and W. Binder, "Execution profiling blueprints," *Software: Practice and Experience*, vol. 42, 09 2012.
- [13] A. Bergel and V. Peña, "Increasing test coverage with hapao," *Science of Computer Programming*, vol. 79, pp. 86 – 100, 2014, experimental Software and Toolkits (EST 4): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT-3 2010). [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0167642312000706>
- [14] D. Holten, "Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 5, pp. 741–748, 2006.
- [15] J. P. Sandoval Alcocer, A. Bergel, S. Ducasse, and M. Denker, "Performance evolution blueprint: Understanding the impact of software evolution on performance," in *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, Sep. 2013, pp. 1–9.
- [16] E. Hart and P. Ross, "Gavel - a new tool for genetic algorithm visualization," *IEEE Transactions on Evolutionary Computation*, vol. 5, no. 4, pp. 335–348, 2001.
- [17] G. Romero, J. J. M. Guervós, P. A. C. Valdivieso, J. G. Castellano, and M. G. Arenas, "Genetic algorithm visualization using self-organizing maps," in *Proceedings of the 7th International Conference on Parallel Problem Solving from Nature*, ser. PPSN VII. Berlin, Heidelberg: Springer-Verlag, 2002, p. 442–451.
- [18] H. Farooq, N. Zakaria, and M. T. Siddique, "An interactive visualization of genetic algorithm on 2-d graph," *Int. J. Softw. Sci. Comput. Intell.*, vol. 4, no. 1, p. 34–54, Jan. 2012. [Online]. Available: <https://doi.org/10.4018/jssci.2012010102>
- [19] H. Farooq and M. T. Siddique, "A comparative study on user interfaces of interactive genetic algorithm," *Procedia Computer Science*, vol. 32, pp. 45 – 52, 2014, the 5th International Conference on Ambient Systems, Networks and Technologies (ANT-2014), the 4th International Conference on Sustainable Energy Information Technology (SEIT-2014). [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S1877050914005961>
- [20] S.-I. Ito, Y. Mitsukura, H. N. Miyamura, T. Saito, and M. Fukumi, *A Visualization of Genetic Algorithm Using the Pseudo-Color*. Berlin, Heidelberg: Springer-Verlag, 2008, p. 444–452. [Online]. Available: https://doi.org/10.1007/978-3-540-69162-4_46
- [21] Y. Tomida, Y. Higo, S. Matsumoto, and S. Kusumoto, "Visualizing code genealogy: How code is evolutionarily fixed in program repair?" in *2019 Working Conference on Software Visualization (VISSOFT)*, 2019, pp. 23–27.