Noname manuscript No.
(will be inserted by the editor)

TestEvoViz: Visualizing Genetically-Based Test Coverage Evolution

Andreina Cota Vidaure · Evelyn Cusi Lopez · Juan Pablo Sandoval Alcocer · Alexandre Bergel

Received: DD Month YEAR / Accepted: DD Month YEAR

Abstract Genetic algorithms are commonly employed 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. Furthermore, evolution processes are most of the time compared solely using the final results, while discarding all the details of the iterations that are themselves important for an adequate tuning.

This paper presents TestEvoViz, a visual technique to introspect genetic algorithm-based test generation processes. TestEvoViz offers the practitioners a visual support to expose the process and decisions made by the generation algorithm. We first present a number of case studies to illustrate the expressiveness of TestEvoViz. We then conducted a user study involving 22 participants including researchers, students and professional software engineers. Participants use our visual approach to analyze, compare and tune test generation algorithm executions. All participants were able to complete the tasks. Our findings show that participants focus more on the visual components that

Andreina Cota Vidaure
SEMANTICS S.R.L., Cochabamba, Bolivia
E-mail: andycotvy@gmail.com

Evelyn Cusi Lopez
SEMANTICS S.R.L., Cochabamba, Bolivia
E-mail: cusi.evelyn@gmail.com

Juan Pablo Sandoval Alcocer
Deparment of Computer Science, School of Engineering,
Pontificia Universidad Católica de Chile, Santiago, Chile
E-mail: juanpablo.sandoval@ing.puc.cl

Alexandre Bergel
University of Chile, Santiago, Chile
RelationalAI, Switzerland
E-mail: alexandre.bergel@me.com

depict information about the test similarity, individuals coverage increments, and the final generation code coverage.

Keywords Automated test generation · Genetic algorithms · Software visualization · Unit testing

1 Introduction

Unit tests have become essential in the software development process. They allow us to verify on a fine-grained level if each unit (i.e., class, method, function) is behaving as expected. Executed automatically on a regular basis as regression tests, they provide a tightly knit safety net for implementing changes and detecting bugs early in the development cycle—but only if various unit tests are available and test all possible usage scenarios.

Unit test can be created manually or automatically. Both ways have their advantages and disadvantages in practice. In a manual test creation, practitioners prepare a set of test scenarios that they believe are needed to be evaluated. Therefore, this activity, besides requiring high manual effort, is limited to a specific perspective of a single developer or tester. Complementary, automatic unit test generation helps to reduce the time to create unit tests and cover scenarios that might be overlooked in the manually crafted unit tests.

A wide spectrum of techniques are commonly employed to generate tests, in particular fuzzing [1], test amplification [2], and genetic algorithms [3, 4]. This paper focuses on supporting the activity of test generation using genetic algorithms. EvoSuite[1] [3] is a popular genetically-based test generation tool. 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 [5]. 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, developers have difficulties understanding the roots of the final generated tests, and the effects of the hyper-parameters in the generation process.

TestEvoViz. We propose TestEvoViz, a visual introspection mechanism for genetically-based test generation. In particular, it helps developers introspect the *whole test suite generation* approach implemented by the EvoSuite tool. *Introspection* refers to the "observation or examination of one's own mental [...] process" and "the act of looking within oneself"[2]. We qualify our visualization an introspection tool since TestEvoViz is meant to support the observation and reflection of the evolution process of the test generation. Figure 1 gives an example of TestEvoViz on a generation of unit tests for the classical class *Stack*, describing a stack data structure. The visualization reads from *top* to

[1] `http://www.evosuite.org`

[2] `https://www.dictionary.com/browse/introspection`

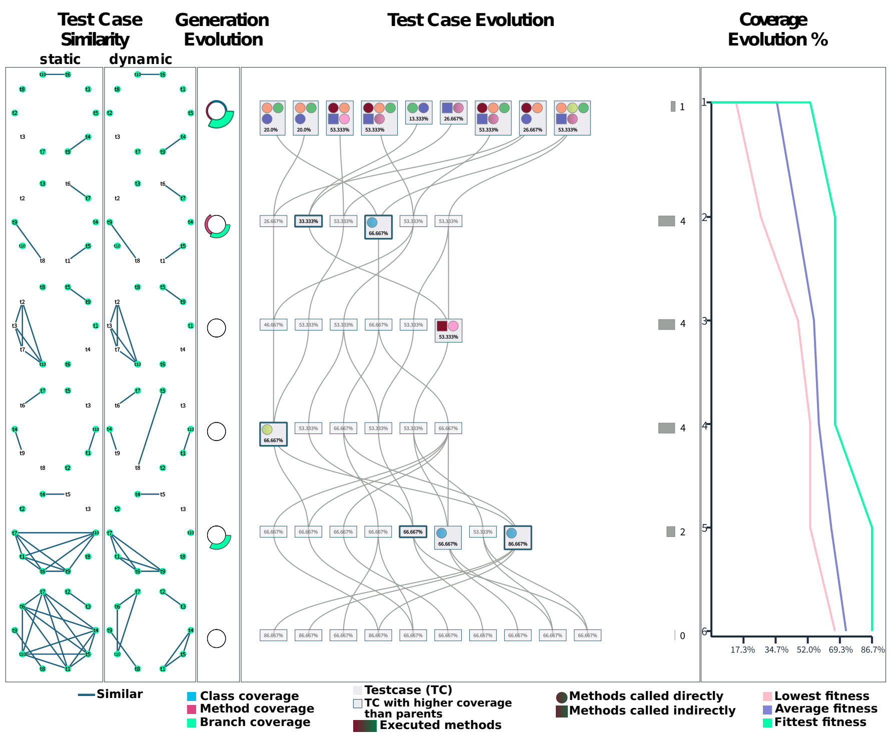


Fig. 1: TestEvoViz - Test generation process for the *Stack* class as an illustrating example. The left-most panel shows the degree of static and dynamic similarity between the unit tests (i.e., individuals) of the evolving population. The next panel, titled *Generation Evolution*, indicates the coverage variation at project level between a given generation and its direct previous generation. The middle panel, titled *Test Case Evolution*, contains generated tests represented as boxes. Links associate each test with its parents. A thick 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 circles are new discovered methods of the tested application that are directly called by unit tests and inner boxes are new discovered methods that are indirectly called by unit tests. Colors represent a method. The right-most panel, called *Coverage Evolution* reports the coverage evolution along generations by rendering the average, lowest and fittest coverage reached in each generation.

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, since there are 6 rows.

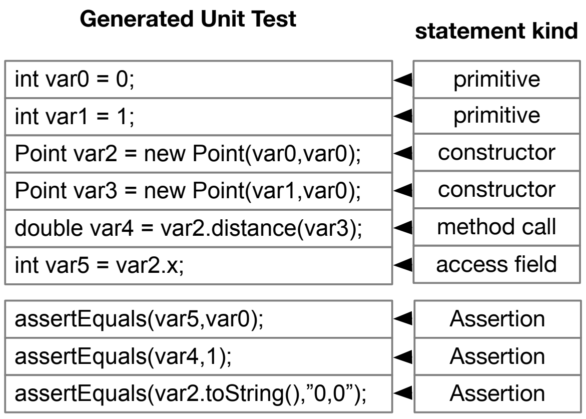
TestEvoViz is composed of four panels, reading from left to right. The first panel titled "Test Case Similarity", located on the left-most of the visualization, represents the static and dynamic similarity between test individuals along the genetic evolution. Second panel indicates the contributions made for each of the 6 iterations. The contribution of each iteration is expressed using a spark circle [6], 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 of test coverage evolution in terms of the best, average, and worse fitness. These curves are relevant for assessing 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 cover 86.7% of the base component under test in the last iteration. TestEvoViz helps developers to understand the impact of the genetic algorithm decisions in the coverage, diversity, and individuals across generations. This information is useful when analyzing, comparing, and tuning test generation processes.

We have applied TestEvoViz to a number of non-trivial examples and conducted a user study with 22 participants. Participants performed three tasks that consist in analyzing, comparing and tuning test generation processes of four real-life software projects. The scope of this study is to comprehend the usage of TestEvoViz from the point of view of a developer, a student, and a researcher. By observing participants behavior, we found that participants focus more on the visual elements that help spot crossover and mutation operations that help increase the population coverage, together with the visual component that shows the similarity between tests. This behavior stems from the relevance for our participants to consider the coverage and the test diversity as important attributes of the generated tests.

Previous work. This article is an extension of a conference paper presented at the eighth IEEE Working Conference on Software Visualization (VISSOFT 2020) [7]. Our previous work is extended in a number of different ways: (i) this article improves our visualization by highlighting the similarity between test individuals along the genetic evolution; (ii) we extend our case studies to illustrate the usefulness of the similarity visualization; (iii) we performed a user study to assess the usability of our visualization approach.

Artifact. This article is accompanied with an artifact, publicly available on `https://github.com/andreina-covi/TestEvoViz`. The artifact contains the video tutorial we used to train the participants, the software TestEvoViz for three different platforms,and the case studies we used in our experiment.

Outline. The paper is structured as follows: Section 2 gives the necessary background to readers unfamiliar with genetically-based test generation; Section 3 describes the TestEvoViz visualization and the introspection mechanism; Section 4 presents some examples that illustrate TestEvoViz in practice; Section 5 presents some real world case studies that highlight the benefits of TestEvoViz; Section 6 summarizes the user study we perform with 22 participants to assess



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

the usefulness of our proposed approach; Section 8 gives an overview of the works related to this paper; Section 9 concludes and presents our future work.

2 Background: genetically-based unit-test generation

2.1 Unit-test generation

A number of techniques have been proposed to automatically generate tests [3, 8, 9, 4, 10]. In this paper, we voluntarily focus on *EvoSuite* [3], a testing tool suite, which uses a genetic algorithm to generate unit tests. In particular, the whole suite approach [11, 12] evolves unit tests 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 and evolving 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 chromosomes. There are four kinds of statements considered by EvoSuite: *primitive* to represent 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 a 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. TestEvoViz does not visualize information related to the assertions statements within the test. However, this point is part of our future work.

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.

2.2 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 hyperpameter values is a process that typically follows a try-and-adjust fashion, and the hyperparameters values may vary depending of the class under test [9, 13].

– *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 high coverage) are generated in an early iteration. Unit test generation is a computationally intensive process and avoiding unnecessary iterations has a significant practical impact [9].
– *Evolution comparisons* – Characterizing execution details of the genetic algorithm is a key aspect to tune hyperparemeters and to determine the stop condition. An evolution, expressed in terms of iterations, involves many operations over the population and its individuals. Comparing different several evolutions and drawing actionable conclusions is therefore crucial.
– *Understanding the roots of the final output* – Test generation tools have different optimization objectives, for instance, maximize a coverage criteria and the mutation score. However, if the output is not as expected, for example, if there are very similar tests - tests without assertions, or tests are asserting methods that do not belong to the target class (indirect tests) [14] - it is difficult to debug and understand the roots of these odd situations.

These four 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.

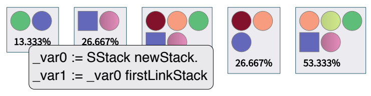
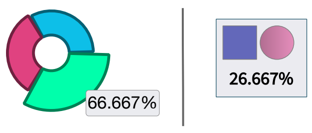
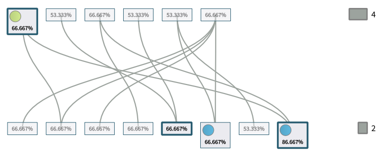
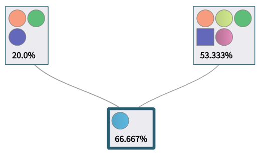
3 TestEvoViz

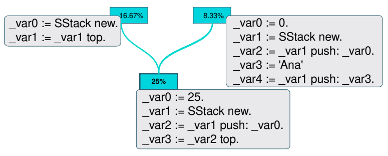
We propose *TestEvoViz*, a visual approach to represent the generation of unit tests using genetic algorithms. *TestEvoViz* visually introspects the algorithm internally to let a practitioner better understand decisions taken by the algorithm. *TestEvoViz* has six main visual components to convey different 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 Figure 1, which illustrates the test generation for the `Stack` class. Table 1 details the relation between the genetic algorithm concepts and the proposed visualization.

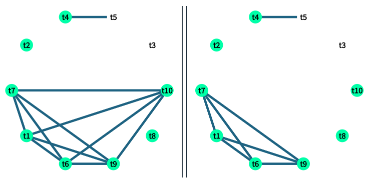
3.1 Data model and introspection

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

Table 1: Mapping genetic algorithm concepts in TestEvoViz

Initialization	Population is composed of N tests, and each test is composed of M statements.	 `_var0 := SStack newStack.` `_var1 := _var0 firstLinkStack`
Fitness	The fitness is given by the branch coverage of each test and is shown at the bottom of each node. Each ring sector in the generation evolution shows the class, method and branch coverage variation of each generation.	
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.	 `_var0 := SStack new.`
		`_var1 := _var1 top.`
		
		`_var0 := 0.`
		`_var1 := SStack new.`
		`_var2 := _var1 push: _var0.`
		`_var3 := 'Ana'`
		`_var4 := _var1 push: _var3.`
		
		`_var0 := 25.`
		`_var1 := SStack new.`
		`_var2 := _var1 push: _var0.`
		`_var3 := _var2 top.`

Similarity	Each node is a test, and two tests are connected if their Jaccard similarity is above a threshold. Left reflects method calls contained in the tests, and right reflects executed methods.	

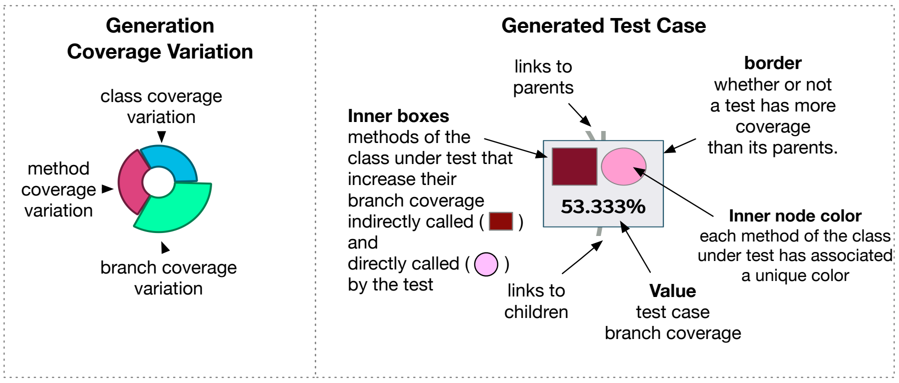

Generation Coverage Variation
class coverage variation
method coverage variation
branch coverage variation
Generated Test Case
links to parents
border
whether or not a test has more coverage than its parents.
Inner boxes
methods of the class under test that increase their branch coverage indirectly called ( ) and directly called ( ) by the test
53.333%
Inner node color
each method of the class under test has associated a unique color
links to children
Value
test case branch coverage

Fig. 3: A spark circle (left side) summarizes the coverage variations of a given generation regarding the previous one. A node glyph (right-side) represents a test and the method that it executes of the class under test.

the number of generations. The initial random population is denoted g_0. Each population g_k consists of m tests $g_k = \{t_0, t_1, \ldots, t_m\}$, 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, depending on whether it results from a crossover operation or not. 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.

3.2 Test case evolution

The middle panel of TestEvoViz (Figure 1) details the unit test evolution along the iterations. Inspired by previous works [15], we use a node-link graph visualization. As in different domains, it is widely used to represent the evolution between entities. A node-link graph representation does not only allow us to show the relation between a test and its evolution, but also group the tests corresponding to the same generation.

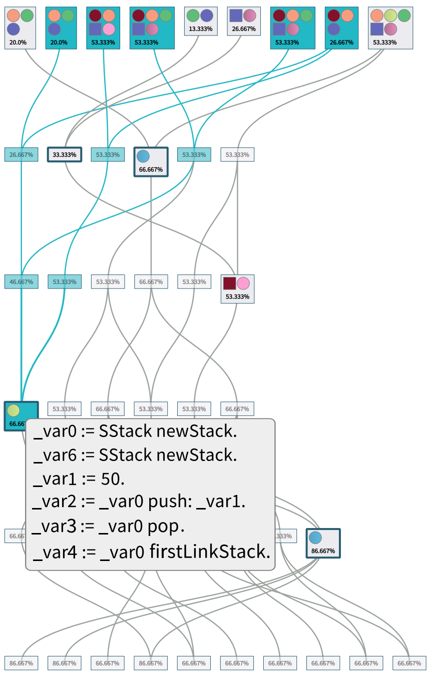
Nodes. Each node represents a test case of a particular generation (t_i, g_j). Tests at a given iteration are horizontally aligned as represented in Figure 1 and Figure 4. In addition, each node is a glyph that displays the methods of the target class and their branch coverage. Let $Bcov(t_i, targetClass)$ 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. The goal is to highlight tests that contribute to the generation goal, in this case, generated test that increase the coverage regarding its parents.
- *Inner nodes* – Each colored inner node represents a method m of the target class that improves its branch coverage regarding the ancestor unit tests $Bcov(t_i, m) > Bcov(t_h, m)$, for all $t_h \in ancestor(t_i, g_j)$. Circular inner nodes represent methods that are called directly from the test case, and Rectangular inner nodes are methods that are called indirectly by the generated tests. To differentiate the methods, each method of the target class has a unique color. The objective is to help developers spot which tests are executing the same methods. 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 $Bcov(t_i, targetClass)$. Since the coverage variation between tests can be small, showing the exact coverage number help developers understand the exact impact of a given mutation and/or crossover in the evolution.

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 depict 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 have more probability to be killed (i.e., not considered or selected to be combined with other unit tests). This depends on the selection strategy used by the algorithm, for instance, the rank selection algorithm assigns more probability to survive to the test that have better fitness function (i.e., higher coverage). However, a test with a low coverage may also survive, this is important, since this test may cover branches that the test with more coverage does not. 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). The number of tests are discarded along each generation is also indicated.



```
_var0 := SStack newStack.
_var6 := SStack newStack.
_var1 := 50.
_var2 := _var0 push: _var1.
_var3 := _var0 pop.
_var4 := _var0 firstLinkStack.
```

Fig. 4: Highlighting ancestors and obtaining source code

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.

3.3 Test case similarity

In the case of unit test, one important aspect in the genetic evolution is the similarity between tests [16, 17]. Understanding the diversity between the individuals of the population represents an opportunity to assess the decision taken by the algorithm and to detect potential redundant generated tests. The test case similarity panel visualizes the similarity between tests along generations. Although there are different alternatives to visualize similarity between elements within a graph, most commonly used are maps and graphs. Our visualization uses network graph for this purpose, previous study shows that it was effective for exploring complex dynamic graphs [18]. In addition, our goal is to provide a general overview of the similarity, since fully understanding how similar two test cases are may require more sophisticated and particular

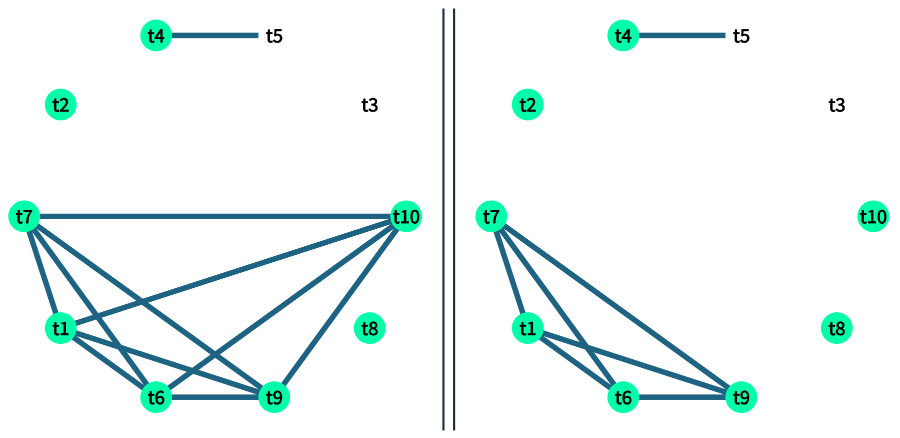


Fig. 5: Static (left) and dynamic (right) similarity between unit tests of a given generation. An edge indicates that the connected tests statically or dynamically call to the same methods. In this example, t_{10} statically calls to the same methods than other tests, but executes a different set of methods, likely the result of using particular argument values.

tools. We do not discard that other alternatives may work similarly or better for this purpose.

For instance, consider Figure 5, it depicts the similarity of the penultimate generation in the Stack example (Figure 1). Each node within both graphs represent the resulting test after six iterations of the genetic algorithm. Each node has a unique number within the graphs, therefore, two nodes with the same number in both graphs represents the same test. Nodes have two background colors: *green* for tests that participate in the creation of the next generation, and *white* for the one that the algorithm discards (e.g., *t3*). Nodes in both graphs are connected according to their similarity.

Static Similarity. Let be $mc(t_i)$ the set of method calls contained within the source code of test t_i. For each pair of test t_i and t_j of a given iteration, we measure their static similarity using the Jaccard index:

$$static\_similarity(t_i, t_j) = \frac{mc(t_i) \cap mc(t_j)}{mc(t_i) \cup mc(t_j)}$$

The Jaccard index, also known as similarity coefficient, is commonly used for measuring the similarity of a sample set. It basically returns the percentage of common elements in both sets. The static similarity measures the ratio between the similar direct method calls between two tests, and the total of distinct method calls done by both tests. We consider that two method calls are similar if they invoke the same method. Note that this metric does not consider the order on which the method calls are done and neither the argument of the receiver. For instance, Figure 5 (left side) shows that there are a number of tests that directly invoke the same methods ($static\_similarity(t_i, t_j) = 1$). Figure 5 (left side) shows that there are five tests that directly call to the same methods ($t_7, t_1, t_6, t_9, t_{10}$), t_5 and t_4 are also statically similar.

Dynamic similarity. To measure the dynamic similarity we detect methods that where executed by a given test. These methods may be called directly or indirectly. Let $em(t_i)$ be the set of executed methods by a test t_i. We compute the dynamic similarity also using the Jaccard index:

$$dynamic\_similarity(t_i, t_j) = \frac{em(t_i) \cap em(t_j)}{em(t_i) \cup em(t_j)}$$

In this case, we are measuring the percentage of methods that were executed by both tests. For instance, Figure 5 (right side) shows that t_7, t_1, t_6 and t_9 cover the same methods during the execution. Note that t_{10} although statically contain method calls to the same methods than t_7, t_1, t_6 and t_9 (as shown with edges in the static similarity, Figure 5 left), during the execution t_{10} calls different methods (as shown with no connecting edges in the dynamic similarity), Figure 5 right.

Two test cases may statically call the same methods, but execute different methods. This may be due to a number of factors, including, the methods are invoked with different arguments, have a different object receiver, or are called in a different order. This may be concluded with further analysis exploring the source code of test t_{10}.

In case both tests do not contain any method call, we consider both tests as similar and assign a similarity of 1. This is an exception to the Jaccard index, because in this situation the divisor of the formula would be zero.

3.4 Generation contribution

The middle panel already shows coverage information of the target class at method level. In addition, inner nodes represent methods that increase their coverage. However, there are other relevant information at the moment to analyze a test execution, in particular, the branch and class coverage because a method can have multiple branches, and a test may execute other classes within the project. Since each generation has an impact on the coverage, we summarize the variation coverage between generations using a spark circle, a circular glyph that shows the variation of multiple metrics. This information may be shown in different ways, but we chose a spark circle due its compact size [19].

The second panel from the left-hand 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 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 of 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. In case that $cov(g_{j-1})$ is zero, we consider that the variation ($\Delta cov(g_j, g_{j-1})$) is zero if $cov(g_j)$ is also zero, and 1 if $cov(g_j)$ is greater than zero.

The execution of a generated test case may cover different methods and classes of a system. Therefore, the height of each ring section is associated with 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 tests 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.

In case that one coverage difference is negative the corresponding spark circle ring will have a bold black border to highlighting this fact. Note that the coverage variation has more probability to be positive because the selection algorithm privileges the tests with more coverage, for instance, if a child has less coverage than the parent, the child has more probability to be discarded. In addition, our EvoSuite implementation applies elitism, which means that the individual with more coverage will survive next to the next generation.

3.5 Coverage evolution

While the middle panel and spark circles show the coverage variation between generations, none of them show the actual evolution of the coverage. The coverage evolution panel shows the evolution of: the fittest unit test per generation (green line), the average of the unit test coverage in a generation (blue line), and the worst unit test per generation (pink line).

4 Examples

This section describes an application of TestEvoViz to introspect the test generation of two classes of the *Pharo* programming language: `Stack` and `DataFrame`.

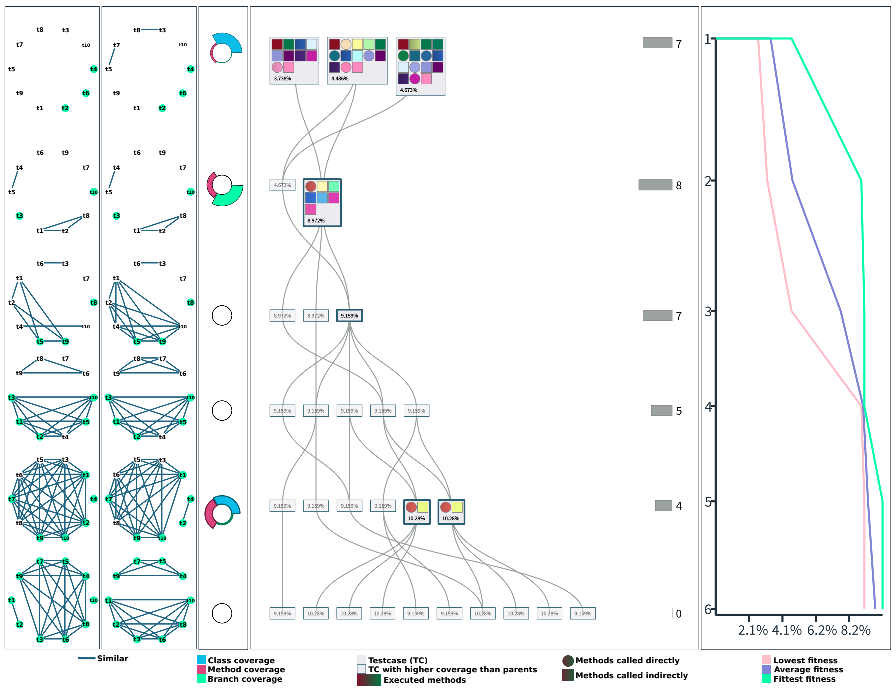


Fig. 6: TestEvoViz on the *DataFrame* class

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 6 depict the results obtained for `Stack` and `DataFrame`, respectively. The following paragraphs detail the test generation as executed by EvoSuite.

Initial Population. The first row of the of middle panel depicts a set of the first randomly generated test. Figure 7 shows the first population of the `Stack` and `DataFrame` example. In the first generation, all tests create an object of the class under analysis, and call a number of method within this class randomly. If the method or constructor have some dependencies (i.e., object or primitives), these are recursively created before calling the randomly selected method. The methods called directly by each test are depicted with circles. For instance, consider the first generation of the `DataFrame` example. There are three tests, the first test (from left to right) directly calls only to one method, because there is only one inner circle inside the test. However, there are eight inner boxes, these represent methods that were called indirectly either by the constructor or the method that is directly called.

In the `Stack` example, we can see that most of the covered methods are called directly, simply looking to the circles. This is mainly because, most of the `Stack`

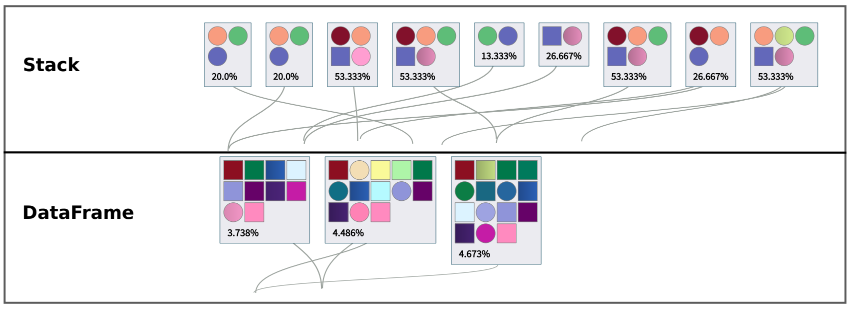


Fig. 7: Initial Population: `DataFrame` and `Stack` example

methods are atomic, and do not call to other methods within the same class. Inner box colors help to detect which tests are calling to the same methods. In the `DataFrame` example, we can see that the test of the first generation directly calls different methods, because the inner circles have different colors. In the other hand, there are couple inner boxes with the same color along the three tests. It means that even thought test directly call different methods, these method indirectly call similar methods.

Crossover and mutation. The middle panel shows the parent-child relation between test cases along the evolution. This relation is depicted by an edge between two nodes. Figure 8 shows the crossover operations and mutation done by the elements in the first generation of the `DataFrame` example.

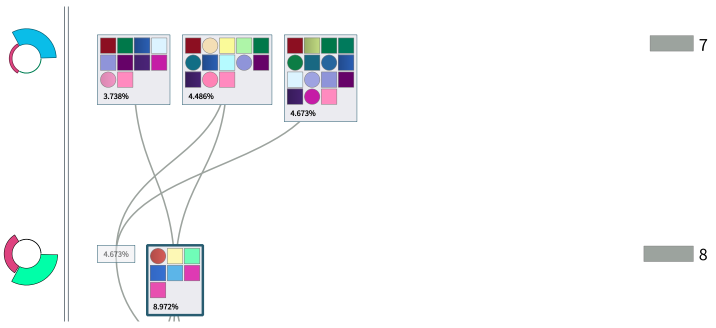


Fig. 8: Crossover and mutation in the first generation: `DataFrame` example

First note that each generation has ten individuals, however, in the case of the `DataFrame` example only three individuals participate in the creation of the final generation. The result of a crossover operation between two tests is represented by the edges, once two tests are merged a mutation is executed over the resulting test. The child of two tests may or not execute branches or methods that were not executed by their parents. This fact is depicted by the inner nodes within the test. Therefore, 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 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. In addition, three of these nodes add a new method call to a test, which is represented with a circle, and these new statements indirectly call different methods in the target class (i.e., rectangular inner nodes).
– *Without inner boxes* – A node without any inner box represents a test case that has a better coverage than its parents, but does 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 to the crossover mechanism. For instance, the third iteration of the DataFrame example has a node that increases its coverage and does not have any new method calls.

Improving generation coverage. Although the inner boxes help to detect which tests in an iteration have a better coverage than their parents. It is possible that they are discovering new branches that may be already covered by the others tests in the same iteration. 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.

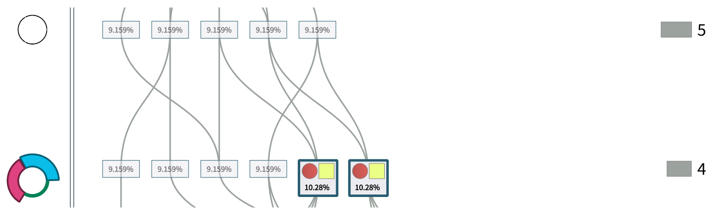


Fig. 9: Increasing method and class coverage: generation 5 - `DataFrame` example

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, consider the fifth generation in the `DataFrame` example (Figure 9). It shows that two tests increase the coverage of their parents, due to a direct call to a method (inner circle), and an indirect call (inner box). In addition, the spark circle shows that in fact new methods were covered, but in addition, a new class was covered. This is indicated by the blue section in the spark circle.

Discarding weak tests. In each generation, the selection algorithm replaces tests with low fitness by evolved tests in a 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 6 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 fourth iteration in Figure 6, because none of the tests of that iteration improves their coverage. However, the lowest coverage increases. This means that even though there was no improvement the algorithm discards test cases with low coverage.

Population diversity. Test case similarity shows the diversity of the test population along the evolution process. For instance, Figure 10 shows the evolution of the static and dynamic similarity of the `DataFrame` example. Note that the tests were becoming more similar along the evolution. In generation four, there were two groups of tests. In generation five, most of the tests have a strong similarity, but the last generation also has two groups of similar tests.

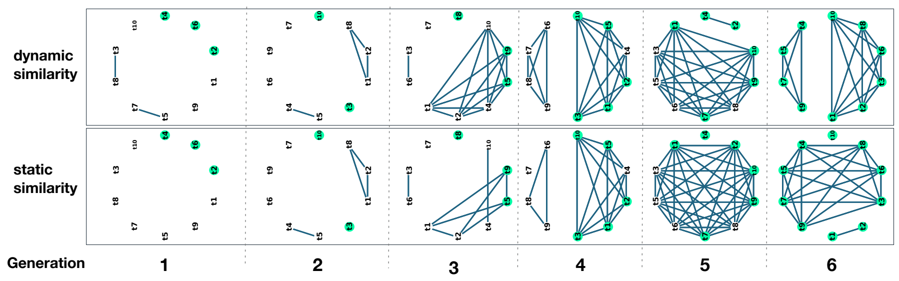


Fig. 10: Test case similarity along generations - `DataFrame` example

Focusing on the generation of tests for the DataFrame example, TestEvoViz shows the following aspects about the generation of tests for DataFrame:

- More indirect methods were invoked by tests in the initial population (Figure 7), when compared to direct method calls;
- Crossover and mutation increase the population coverage in generation two (Figure 8) and generation five (Figure 9);

– The similarity between tests converge to two groups of tests that invoke similar methods, and three groups that directly invoke the same methods (Figure 10).

5 Case Studies

This section presents two case studies on which we use TestEvoViz to analyze the test generation of two *Pharo* projects. For each one of these projects, we visualize the test generation process using a different set of hyperparameters and describe the effects through TestEvoViz of these in the generation process. We select these two projects because both are well know not only in the Pharo community but in research and in general in the industry. Both projects have a similar implementation in different programming languages and are used in different domains.

For each case study, we generate tests for a given class using different parameter configurations, then we use TestEvoViz to highlight the effects of the parameter variation within the generation process. In particular, we focus on three parameters: number of statements, population size, and mutation rate.

Adequately selecting the hyperparameters is a complex task, as there is not a unique best configuration for all kind of applications. Furthermore, it is often necessary to tune the parameters according to a specific problem domain [9, 13]. In our cases study, we use a set of parameters that help us illustrate through our visualization the effects of the parameter configuration. Although we initially based our configuration with EvoSuite default values (i.e., mutation rate) and a previous study of hyperparameter tuning [9], we choose relatively small values for the population size and number of generations for didactic purposes.

5.1 Regex

Regex is a standard *Pharo* library to parse and match regular 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 four configurations (Table 2). Figure 11 gives the result of running the algorithm with the previous configuration. As we see, most of the methods and branches are covered at the beginning of the first iteration. In the next generations there are new test cases that cover more branches than their parents, however, the spark circles show that there were already other test individuals that cover these branches. In the last generation, the test individuals that survive have a similar coverage: this information is represented in the right panel where the lowest, average, and highest coverage of the population are close. After five iterations the tests with the highest branch coverage are 19.78%. Finally, Figure 11 also shows that there are a

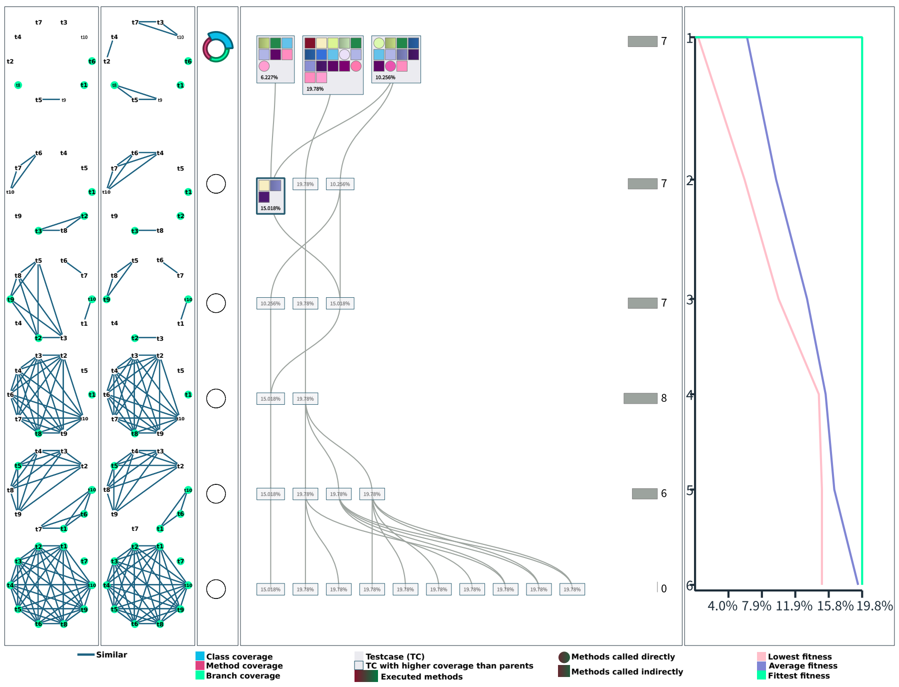


Fig. 11: TestEvoViz – Regex project (Base Configuration); $number\_of\_statements = 5$; $number\_of\_iterations = 5$; $selection\_algorithm = rank\_selection$; $population\_size = 10$; and $mutation\_rate = 1/3$

number of tests with similar method calls, and that all of the generated tests cover similar methods.

Table 2: Regex Analyzed Configurations

Parameters	Base Conf.	Conf. 1	Conf. 2	Conf. 3
Number of Statements	5	3	5	5
Number of Generations	5	5	5	5
Selection Algorithm	Rank	Rank	Rank	Rank
Population Size	10	10	20	10
Mutation Rate	1/3	1/3	1/3	2/3

Number of statements. Figure 12 depicts the generation process using the same base configuration, with the exception that this time we reduce the number of statements from five to three. Figure 12 shows that in contrast to the baseline (Figure 11), the population achieves the highest coverage in the last generation. The green section of the spark circle in this generation shows

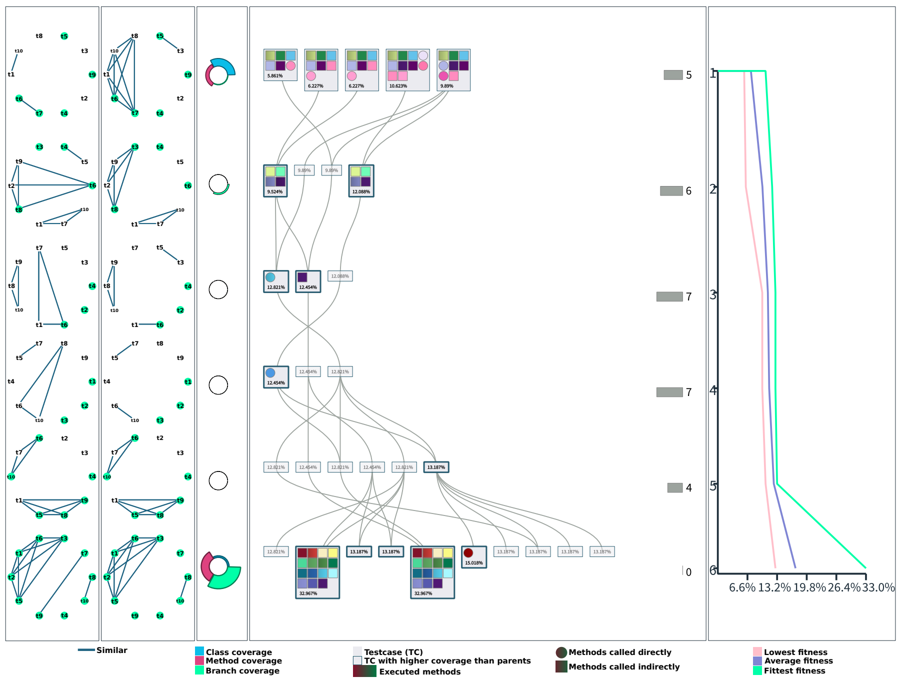


Fig. 12: TestEvoViz – Regex project (Configuration 1); *number_of_statements* = 3; *number_of_iterations* = 5; *selection_algorithm* = *rank_selection*; *population_size* = 10; and *mutation_rate* = 1/3

that the resulting test individuals cover new branches and methods of the target class. Different from the baseline, the individuals are more diverse in terms of method calls, but half of the individuals still cover similar methods and have similar method calls.

Along the evolution eleven tests have more coverage than their ancestors, notable when searching for nodes with a thick border. This particular visualization shows that the crossover operations between individuals with less statements achieve a higher coverage compared to the baseline. With this configuration, the best generated test case covers 33% branches of the target class, which is more than the baseline (19.78%).

Population size. Figure 13 depicts the generation process using the same base configuration, with the exception that this time we increase the population size from 10 to 20. The fourth generation in Figure 13 contains two tests that cover new branches regarding their parents. These tests contribute to increasing the branch coverage of the population, as indicated by the spark circle in the fourth generation. Similarly to the baseline, the visualization shows that there are few tests that have better coverage than their ancestors. But in this

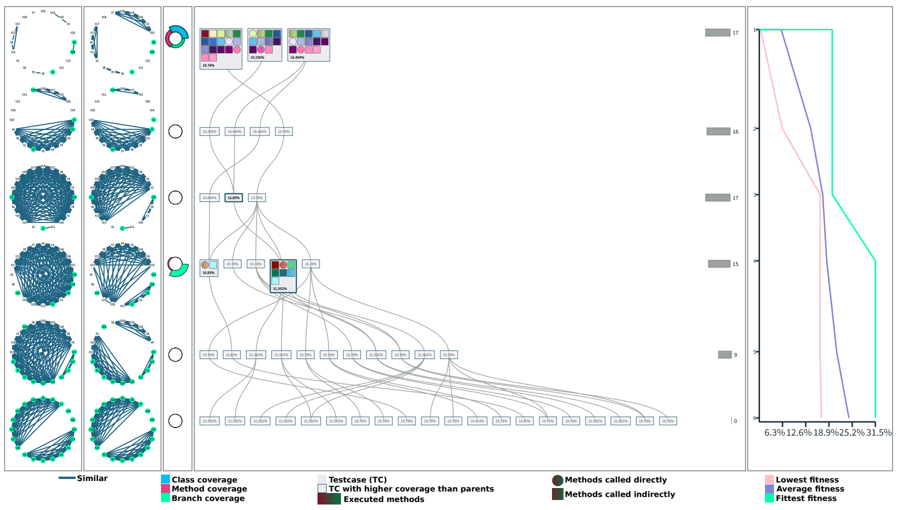


Fig. 13: TestEvoViz – Regex project (Configuration 2); $number\_of\_statements = 5$; $number\_of\_iterations = 5$; $selection\_algorithm = rank\_selection$; $population\_size = 20$; and $mutation\_rate = 1/3$

case, the algorithm found a new test case which got better coverage than the baseline. However, there are two groups in the last generation that have similar method calls (statically) and cover similar methods (dynamically).

Mutation rate. Figure 14 gives the generation process using the base configuration, but this time increasing the mutation rate from 1/3 to 2/3. Note that this time, the coverage of the last population is 35.897%, which is greater than the one obtained with previous configurations. In this case, an individual of the second generation increases its coverage, then in the following generations the remaining individuals progressively increase their coverage, however, no new branches were discovered after generation two. The similarity panel shows that in the fourth generation most of the individuals contain and execute similar methods. However, in the last generation only half have covered similar methods.

5.2 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 four configurations (Table 3). Using a greater number of generations and statements has the effect of producing a larger visualization. Figure 15 shows the test evolution process for the class `NeoJSONObjectMapping`.

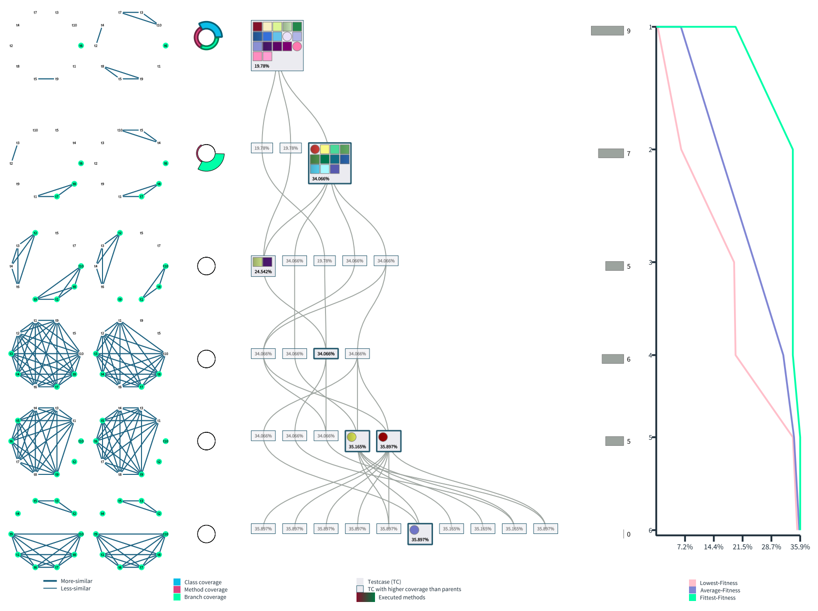


Fig. 14: TestEvoViz – Regex project (Configuration 3); *number_of_statements* = 5; *number_of_iterations* = 5; *selection_algorithm* = *rank_selection*; *population_size* = 10; and *mutation_rate* = 2/3

Table 3: Json Analyzed Configurations

Parameters	Base Conf.	Conf. 1	Conf. 2	Conf. 3
Number of Statements	10	20	10	10
Number of Generations	10	10	10	10
Selection Algorithm	Rank	Rank	Rank	Rank
Population Size	20	20	30	20
Mutation Rate	1/3	1/3	1/3	2/3

As we see, the population coverage slowly increases along generations. The genetic algorithm is discarding tests with a lower coverage, and in the last version, the coverage of the population is similar. This fact is shown through the coverage evolution panel. Spark circles show that new branches were discovered in generation three and five, and a new method and a new class was executed by a test in the last generation.

In Figure 15, the similarity panel shows that at the beginning of the evolution the tests cover different methods, but along the evolution, tests are becoming dynamically and statically similar. This fact is due to the number of statements configuration, since the number of statements is 10 and the num-

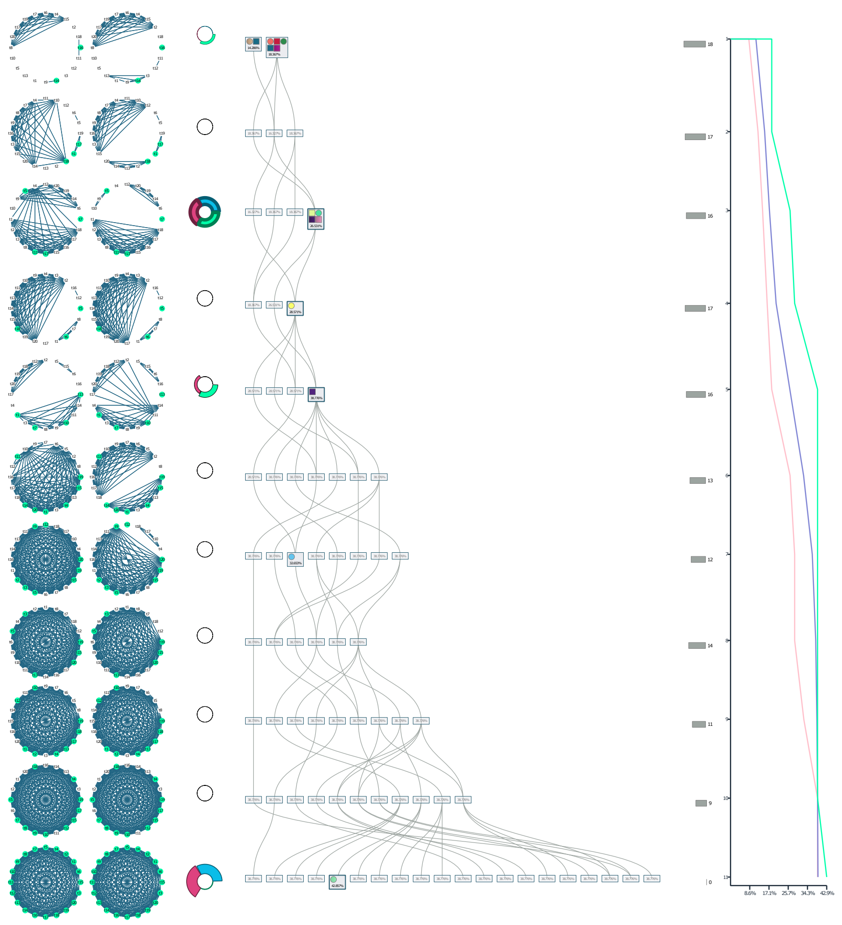


Fig. 15: TestEvoViz – NeoJSON project (Base Configuration); $number\_of\_statements = 10$; $number\_of\_iterations = 10$; $selection\_algorithm = rank\_selection$; $population\_size = 20$; and $mutation\_rate = 1/3$

ber of class methods is 16, there is a higher probability of calling the same methods.

Number of statements. Figure 16 depicts the generation process using the same base configuration, with the exception that this time we increase the number of statements from ten to twenty. Figure 16 shows the visualization of this change. First, we notice that (i) the last generation has a lesser coverage than the baseline, and (ii) most of the branches are discovered in the first generation. The similarity panel shows that due to the high number of statements, tests tend to call to the same methods since the third generation.

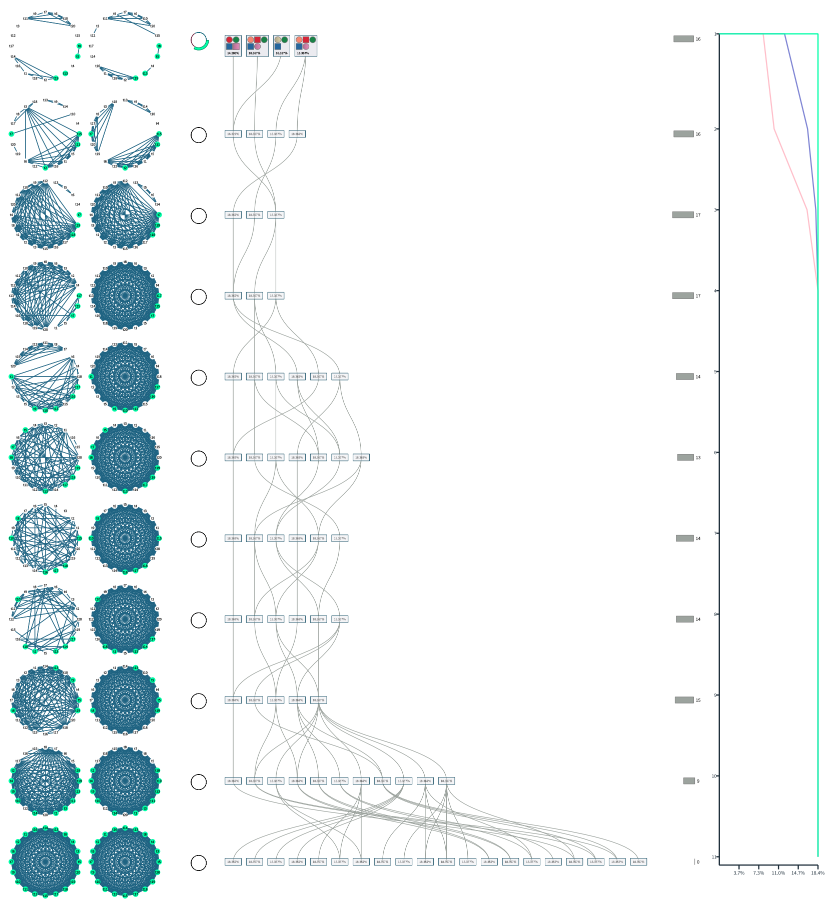


Fig. 16: TestEvoViz – NeoJSON project (Configuration 1); *number_of_statements* = 20; *number_of_iterations* = 10; *selection_algorithm* = *rank_selection*; *population_size* = 20; and *mutation_rate* = 1/3

Therefore, we concluded that in this particular case, increasing the number of statements did not help the generation process.

Population size. Figure 17 depicts the generation process using the same base configuration, but uses a population size of 30 instead of 20. The coverage of the population evolves from 20 to 40, similar to the baseline. In this case, spark circles show that generation 8, 9 and 11 discover new branches and methods. The similarity between tests varied during the evolution, but in the last generation most tests cover similar methods.

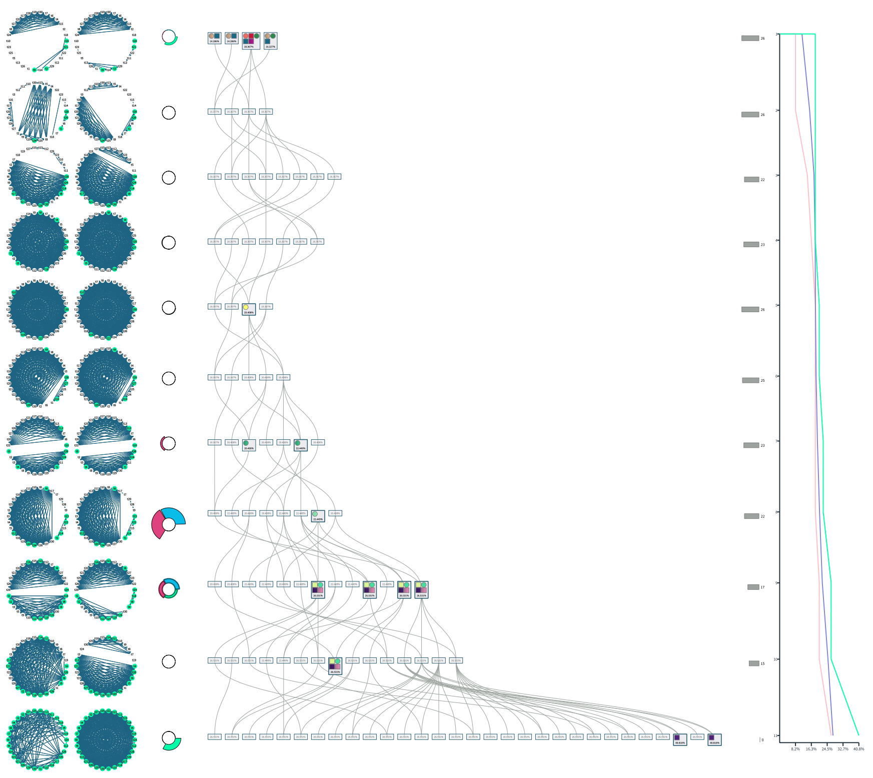


Fig. 17: TestEvoViz – NeoJSON project (Configuration 2); *number_of_statements* = 10; *number_of_iterations* = 10; *selection_algorithm* = *rank_selection*; *population_size* = 30; and *mutation_rate* = 1/3

Mutation rate. Figure 18 details the generation process using the base configuration, but this time increasing the mutation rate from 1/3 to 2/3. Figure 18 shows that the coverage evolution is similar to the baseline. The similarity between tests is lower in the first five generations, and new branches were discovered in the fourth and seventh generations. The coverage of the last population is similar to the baseline.

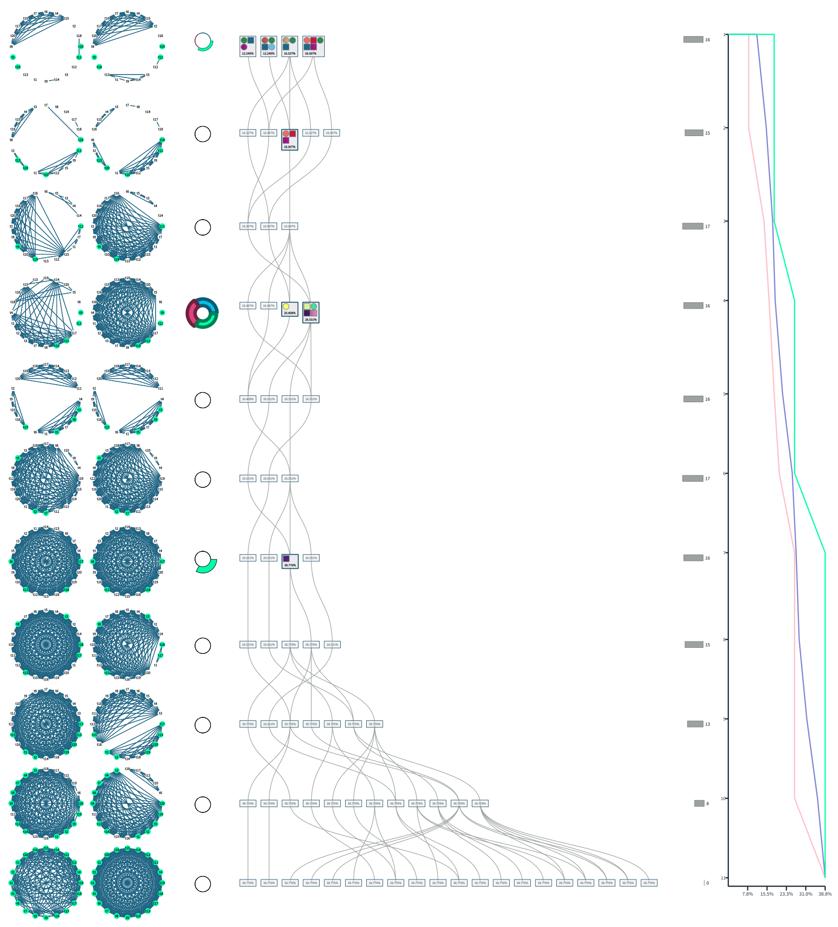


Fig. 18: TestEvoViz – NeoJSON project (Configuration 3); $number\_of\_statements = 10$; $number\_of\_iterations = 10$; $selection\_algorithm = rank\_selection$; $population\_size = 20$; and $mutation\_rate = 2/3$

6 User Study

In this section we describe the research questions and the methodology we follow to conduct our study.

6.1 Research Questions

The overall goal of this study is to examine the usage of TestEvoViz in the context of analyzing, comparing and tuning genetic algorithm based test generation processes. Therefore, we state our main research question as follows:

How and how well do developers use TestEvoViz to analyze, compare, and tune test generation evolution process?

The first part of the study is about analyzing developer perceptions of usability and cognitive load of using our visualization. In addition, identify problems and advantages they have while using TestEvoViz. Hence, the first part of our study address the following research questions:

– **RQ.1** *What are developers' usability perceptions of TestEvoViz?*
– **RQ.2** *What are developers' cognitive load perceptions of TestEvoViz?*

The second part of the study is about understanding how developers use the proposed visualization to analyze, compare and tune the hyper parameters needed by the test generation algorithm. Hence, our third and fourth research questions are:

– **RQ.3** *How do developers use TestEvoViz to analyze and compare test generation processes?*
– **RQ.4** *How do developers use TestEvoViz to tune hyper parameters?*

6.2 Experimental Setup

6.2.1 Methodology Overview

To answer our research questions, we propose a methodology structured along six stages:

1. *Project under Study.* We select a number of projects over which participants will perform the experiment.
2. *Video Tutorials & Training Session.* We made a video tutorial and designed a training session in which participants use the visualization to answer a number of questions in order to get familiar with the tool.
3. *Task Design.* We designed tasks focused on three dimensions: analysis, comparison, and tuning test generation processes.
4. *Pilot.* We perform a pilot in order to find issues and improve the tutorial, training session, and tasks design.
5. *Participant Recruitment.* We recruited 22 participants to participate in our study with different backgrounds in academia and industry.

6. *Work Session & Data Collection.* We design a work session for each participant and define the instruments we use to collect the necessary data to answer our research questions.

The remainder of this section elaborates on the stages described above.

6.2.2 Video Tutorial & Training Session

Before carrying out the training session, each participant receives by email a survey about demography, a video tutorial, and a set of instructions to download and run the artifacts needed for the experiment. During the training session, a participant has to generate unit tests for a `Stack` class, which we consider as a simple toy example. This small exercises requires the participant to interpret the visualization and remember the meaning of each component. In addition to its pedagogical purpose, this training sessions serves to evaluate if the participants really understand the tutorial and gives them a chance to ask for clarifications.

While participants were reviewing and interacting with TestEvoViz, we clarified the doubts and questions that they asked us regarding particular components. After the clarifications, all participants felt confident to understand all visual components within the visualization.

6.2.3 Tasks

We define three tasks in order to evaluate our proposed visualization in three dimensions: analyze, compare, and tune test generation processes. Table 4 describes each one of these three tasks, and their rationale. While all the tasks focus on answering our research questions, the task *T3* mostly focuses on answering the research question *RQ4*.

6.2.4 Pilot

We perform a pilot with a software engineer that develops and maintains a genetically based generator tool for Pharo. The pilot helped us: (1) clarify our questionnaire; (2) reduce the tasks workload, since the pilot took two hours longer than we initially hoped. Before the pilot task we asked participants to describe the important facts they see in the test evolution of six generations.

We reduce the workload by reducing these tasks to analyze only three generations. But, we let participants select three generations which they consider more interesting to analyze than others. In a similar fashion, task three compared the evolution of five pairs of different configurations. We reduced the task to compare only two pairs of configurations. After these adjustments, we conducted a second pilot with a different engineer with experience in test generation. The time needed to complete the task was 45 minutes, which also helped reduce the fatigue effect between tasks.

Table 4: Tasks

T1	**Concern:** Analysis **Description:** Choose three generations that you consider more interesting and describe the most important fact you saw in these generations. **Rationale:** The goal of this task is to evaluate how developers use the visualization to assess the generation process, and understand how they relate their conclusions with the visual components. In particular, we are interested in understanding which aspects of the evolution process participants consider in their conclusions and which aspects are not mentioned. **Associated Research Questions:** RQ1 & RQ2 & RQ3
T2	**Concern:** Comparison **Description:** Compare three different generation processes (each with different hyper-parameters), summarize the differences and similarities between them. **Rationale:** We provide participants three visualizations, each with a different configuration. This task is about understanding the participant decision process about the parameters in the configuration. In particular, we are interested in determining which factor participants are considering when choosing one configuration over others. Subsequently, the participants have to describe the similarities and differences. Finally, we ask participants to choose which configuration is more suitable. **Associated Research Questions:** RQ1 & RQ2 & RQ3
T3	**Concern:** Hyperparameter tuning **Description:** Configure the hyperparameters in order to improve the generated tests of a given target class. **Rationale:** We provide each participants three well know classes for which they need configure the hyper-parameters and generate tests. Participants have to select and tune the hyper-parameters until they were satisfied with the generated tests. **Associated Research Questions:** RQ1, RQ2, & RQ4

6.2.5 Projects under Study

To keep the task manageable, we use a code base that is relatively known to all the participants of the experiment, since these projects are part of the Pharo core. For task *T1*, we have a basket of four projects: *NEOJSON*, a JSON parsing library. *Regex*, a regular expression library, *DataFrame*, a popular data structure, *Box2DLite*, a small 2D physics engine. Each participant performs task *T1* and task *T2* on a different project randomly assigned, in order to not favor any project or task.

In case of task *T3*, hyper parameter tuning, participants need to have strong knowledge about the class under test in order to assess the quality of the generated tests. In addition, participants will generate tests multiple times with different parameters, therefore, the project under analysis has to be relatively small in order to reduce the overhead needed for the generator tool. Otherwise, participants will spend more time waiting for the tool than analyzing the generation process. For this reason, and inspired in previous works, for this task, we considered three popular classes as classes under tests: ATM, Rectangle, and Vector. The first target class is taken from the Pharo core package, the second is an implementation of the main functionalities of ATM, and the third is taken from the PolyMath project.

6.2.6 Participants

We sent an open invitation to the Pharo developer community and the authors university student and alumni mailing list. The Pharo community is composed of academics, researchers, developers and student from different countries. In addition, we invited researchers that work in test generation and software testing, searching in them in conferences in the area.

In total, we have 22 volunteers that participate in our experiment. We picked the participants according to their expertise in software testing and programming experience. Six PhD students, two postdoctoral researchers, six professional engineers, one associated professor, one university lecturer, a master student, and four undergrad students. Their programming experience ranges from 1 to 30 years. 19 of them have 5 years of experience or more, and only 3 of them have less than 5 years of experience. Note that the undergrad students that participate in the experiment, already work in the software industry, in parallel to their studies. Twelve of them are familiar with test generator tools, and all of them have experience in unit tests.

Due the time constraints not all participants perform all the three tasks, we balance the effort and we ensure that each task was performed by 14 participants. Table 5 details the experience and the task each participant perform during the experiment. Note that all participants participate in the video tutorial and in the learning session. These 22 participants are different, to the ones that perform the pilot.

6.2.7 Work session & Data Collection

Figure 19 gives an overview of the work session and the data collection. The session of each participant is structured as follows:

1. *Demographic Questions* – We first ask the participants to indicate their current occupation, programming, testing, and test generation tools experience.
2. *Video Tutorial & Training Session* – All participants review the video tutorial, and perform the training session on which analyze the test generation process of one of the projects under study assigned randomly.
3. *Task Execution* – Due time constraints all participants could not execute the three tasks. For this reason, each participant was assigned to perform one or two tasks. Table 5 gives the task assigment of each participant. Each task was performed by exactly 14 participants. For task `T1` and `T2`, we assign randomly a project under analysis, balancing the assigment across participants. In case of task `T3`, participants perform the activity for the three target classes, but we randomly assign the order.

Table 5: Participants (P.E. = Programming Experience (years); T.E.= Testing Experience (years); T.G. = Familiar with Test Generation Tools (Yes/No); T1, T2, T3 = Participation in a particular task)

ID	Position	Research Area	P.E.	T.E.	T.G.	T1	T2	T3
P1	Postdoctoral Researcher	Software Testing	20	15	✓	✓	✓	✗
P2	Postdoctoral Researcher	Software Engineering	11	1	✗	✓	✓	✓
P3	Senior Professional Engineer (PhD)	Software Engineering	22	14	✗	✓	✓	✓
P4	PhD Student	Software Testing	14	2	✓	✓	✓	✗
P5	Senior Professional Engineer (PhD)	Software Testing	6.5	6	✓	✓	✓	✓
P6	Associate Professor	Software Evolution	15	13	✓	✓	✓	✗
P7	Lecturer (PhD)	Software Testing	18	14	✓	✓	✓	✗
P8	Assistant Project Scientist (PhD)	Software Engineering	12	6	✓	✓	✓	✗
P9	PhD Student	Test Amplification	8	3	✓	✓	✓	✗
P10	Master	Software Engineering	6	2	✗	✓	✓	✗
P11	Senior Professional Engineer	Software Engineering	12	10	✗	✓	✓	✓
P12	Professional Engineer (PhD)	Electric Engineering	10	5	✗	✓	✓	✓
P13	PhD Student	Software Evolution	2	1	✗	✓	✓	✗
P14	PhD Student	Software Evolution	8	7	✗	✓	✓	✓
P15	Student	Computer Science	1	1	✗	✗	✗	✓
P16	Student	Computer Science	4	1	✗	✗	✗	✓
P17	Student	Computer Science	5	3	✓	✗	✗	✓
P18	Student	Computer Science	5	2	✓	✗	✗	✓
P19	Senior Professional Engineer	Software Engineering	25	10	✗	✗	✗	✓
P20	Professional Engineer	Software Engineering	3.5	2.5	✓	✗	✗	✓
P21	PhD Student	Software Engineering	8	6	✓	✗	✗	✓
P22	Senior Professional Engineer	Software Engineering	30	28	✓	✗	✗	✓

4. *NASA Task Load Index (TLX)* – After completing each task participants fills a NASA TLX [3] to detail their perceptions of the cognitive load of each one of the tasks [20].
5. *System Usability Scale (SUS) Form* – After completing all their assigned tasks each participant fill a usability using the SUS form [4], to evaluate the usability of the proposed visualization [21].
6. *Feedback* – Finally, each participant verbally provides the advantages and disadvantages, improvement suggestions, and other commnet that they have about TestEvoViz.

We monitor the completion of the tasks and record participant's screen during all the experiments. Furthermore, we invite the participants to speak out on their thoughts, questions, and indications about their progress while carrying out the tasks. For this last point, we previously ask for participant consent. The answers of all participants are anonymous and available online [5].

[3] https://humansystems.arc.nasa.gov/groups/TLX/

[4] https://www.usability.gov/how-to-and-tools/methods/system-usability-scale.html

[5] https://bit.ly/3xHh6Yw

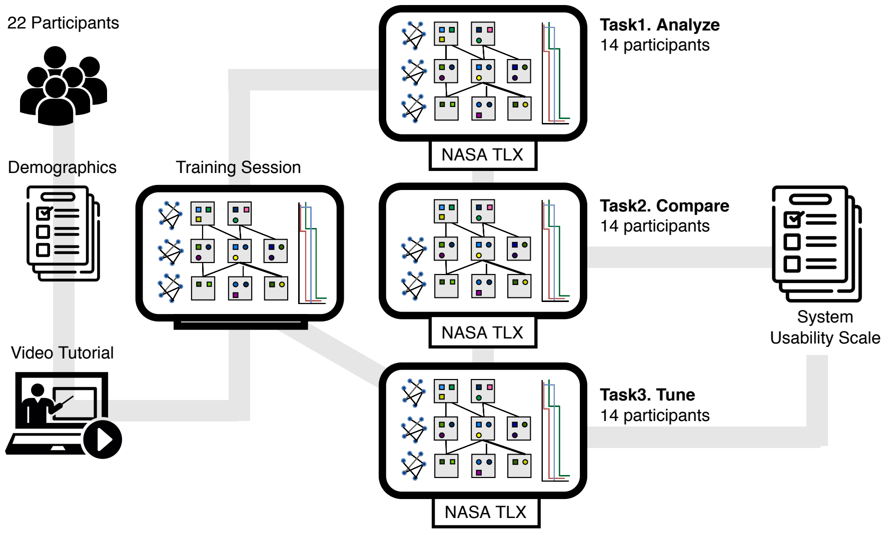


Fig. 19: Work Session & Data Collection Overview

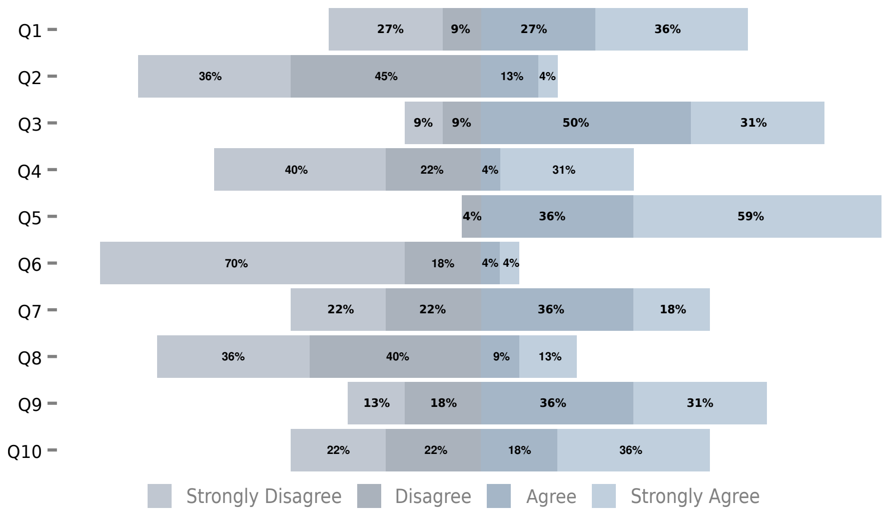
6.3 Results

6.3.1 RQ.1 What are developers usability perceptions of TestEvoViz?

Each participant uses a Likert scale to rate each one of the affirmations done in the questionnaire. Figure 20 lists the questions and participant answers in the system usability scale form (SUS). We sum up the score for each participant, and then multiplied the score by 2.5 to convert the original scores of 0-40 to 0-100, as advised in the original description of the SUS form [21]. Participants' usability score ranges from 42.5 to 100, with a median of 70.

In total, 18 participants agreed TestEvoViz was easy to use. Three participants (P1, P8, P9) said that it was not as easy and P7 found TestEvoViz complex. Regarding confidence, 19 of the 22 participants felt confident or partially confident using it. P7 did not feel confident because the participant doubted the usefulness of the information shown in the visualization tool, P18 said the middle panel (TestCase Evolution) was hard to understand because it does not have a description of the components meaning (it is easy to forget its meaning).

On the other hand, two participants (P2, P8) manifested that they would not use it frequently, since they do not use test generation techniques frequently either, and P7 said: *"I would not say frequently, perhaps I'd use it sometimes"*. Other perceptions of the participants related to the tool were: P13 - *"I had problems to understand the tool, but even I'm not very familiarized with genetic algorithm and test generation, I could use it and I think this speaks well of the tool"*. P16 - *"The tool helps me to see if it is of any use to*



Q1 - I thought that I would like to use TestEvoViz frequently
Q2 - I found TestEvoViz unnecesarily complex
Q3 - I thought TestEvoViz was easy to use
Q4 - I think that I would need the support of a technical person to be able to use TestEvoViz
Q5 - I found the various functions in TestEvoViz were well integrated
Q6 - I thought there was too much inconsistency in TestEvoViz
Q7 - I would imagine that most people would learn to use TestEvoViz very quickly
Q8 - I found TestEvoViz very cumbersome (awkward) to use
Q9 - I felt very confident using TestEvoViz
Q10 - I needed to learn a lot of things before I could get going with TestEvoViz

Fig. 20: SUS Scale Results. The system usability scale (SUS) consists of a ten item questionnaire with five response options for respondents; from Strongly disagree to Strongly agree (in a 5 point likert scale). This figure summarizes the answers of 22 participants about the usability of TestEvoViz.

change these things, for example the number of generations, if it is of any use to increase or not“. P6 - *“The times I've used the generated tests, I've always asked which was the similarity degree between the tests and many times I didn't have it clear. It means, I generated a lot of tests with EvoSuite, a big amount, and I used to say to myself that I don't see the diversity between the generated tests, then I think this tool lets me see the panorama in those cases*“.

Overall, eight participants' scores were equal or greater than 85, ten participants scored from 60 to 75, two participants scored 47,5, and one participant scored 55, and the other 42,5.

Conclusions. According the comments of the 22 participants about the tool, we can conclude:

– Most of the participants agreed the tool was easy to use and felt confident using it;

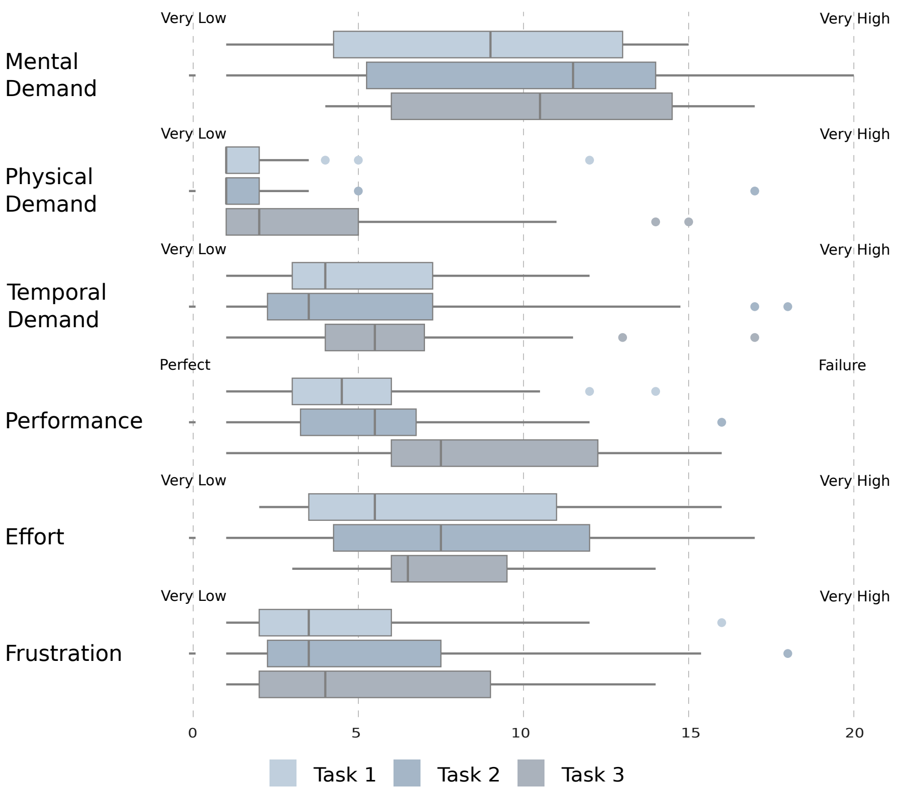


Fig. 21: NASA-TLX Cognitive Effort Summary. Each color correspond to a task

– Considering the threshold of 68, commonly used to qualify usability systems [21], we can claim that the usability of TestEvoViz is acceptable.

6.3.2 RQ.2 What are developers cognitive load perceptions of TestEvoViz?

Figure 21 shows that participants perceive more physical demand during task T3. This is mainly because, task `T3` consists in tuning hyperparameter values which implicitly requires analyzing and comparing visualizations to understand the effect of different parameters. Tasks `T1` and `T2` are less constrained since participants have to characterize and compare the test generation process without having to modify any values. Six and seven participants score this task higher than the average.

We do not set any time restrictions for any of the three tasks. Some participants completed the task faster than the other participants. The range of the time for completing the task `T1` was from 5 minutes to 37 minutes. For task `T2` the range was from 6 minutes to 30 minutes. And for task `T3` was from 20 minutes to 87 minutes.

The wide time gap in performance for task T3 is because participants explore different parameter configurations until they get satisfied with the generated tests. 11 participants perceived their performance from good to perfect in the three tasks. But some participants (two in the tasks T1, T2, and three in the task T3) felt kind of frustration while doing the tasks because they forgot the usefulness of some components, or they could not get a higher coverage.

Regarding, the cognitive effort perceptions in mental demand, temporal demand, effort, and frustration. Figure 21 show small differences between mean values, however, there is not a strong difference between tasks. In these particular, the box plots in these dimensions are overlapping each other without any clear difference.

Conclusions. According the answers of the participants about their cognitive load perceptions, we argue that:

– Participants perceive task T3 as more physical demanding than task T1 and T2, since the activity of tuning hyperparameters requires also comparing and analyzing the resulting visualizations;
– Participants have less confidence in their performance during task T3, since they were unable to achieve higher coverage during the parameter tuning;
– All the tasks have similar perceptions about mental demand, temporal demand, effort, and frustration.

6.3.3 RQ.3 How do developers use TestEvoViz to analyze and compare test generation processes?

Task 1: Analyze. This task is about selecting 3 generations and detailing the facts during the evolution process that participants found the most important. Most selected generations were the ones that increased the branch coverage or presented more colorful inner boxes. However, P3 selected the generation that contained a test with many descendants, P4 selected the first generation since most of the branches are discovered in this generation. Six participants (P2-P6, P9) noticed the similarity between the tests in these generations. Participants were curious about the generated code (using the popup), mainly to analyze the similarity between tests. All of them prioritized the tests that contain inner nodes, since they represent individuals that discover new branches regarding their ancestors. Three participants (P2, P3, P9) used the interactions to highlight the ancestors of a given test in order to understand why some tests were similar. Spark circles and the branch evolution chart were only used to confirm that a given test or generation contributes to the coverage increment.

Task 2: Comparison. In this task, participants analyzed two visualizations, each one generated with a different set of hyperparameters. Participants had to highlight the most relevant differences between two evolution processes, select the one they consider most useful, and justify their selection. All participants essentially focused on the final branch coverage value as a principal attribute for their final decision. Eight of 14 participants (P1, P2, P4-P9) highlighted the importance of the test similarity in the final generation (note that we had 22 participants, however each task was carried out by 14 participants to avoid overloading the participants). Consequently, participants highlighted the generated tests that have more coverage than their parents, and analyzed how these tests contribute to achieving a higher coverage in future generations. Only three participants (P6, P8, P9) related the differences among values in the configurations and their impact in the generation process. However, P2, P3 and P7 expressed their expectancy to get better results on visualizations configured with a larger population size.

Conclusions. By observing the 14 participants completing the tasks $T1$ and $T2$ (which are related to RQ.3), we make the following claims:

– All participants paid attention to the final branch coverage value as a principal attribute for their final decision;
– 8 of 14 participants highlighted the importance of the test similarity in the final generation.

6.3.4 RQ.4 How do developers use TestEvoViz to tune hyperparameters?

Task 3: Tuning. The task T3 consists in tuning the hyperparameters of three different classes (ATM, Rectangle, and PMVector) and selecting a configuration that generates better tests according to their personal criteria. A script with a default setting of the hyperparameters values was given, for each class, and the participants were able to change these values, execute the script, and watch the effect of those changes in the visualization tool. The participants could modify the values as many times they deemed necessary.

Figure 22 shows a visual summary of three participant sessions, in which the contrast of the patterns are notorious (figures in Appendix show the visual summary of all the participant sessions). Each row represents a participant session, in particular, (i) the hyperparameters the participants modify and (ii) the visual components they use to analyze the test generation process. Each participant tunes hyperparameters for the three classes under study. During the session participants generated tests multiple times with different parameters, each test generation execution is represented with a dotted line.

Each row is split into two parts with a bold line. The top part displays the visual components that participants analyze during the session. Each visual component is associated with a color and the component name is on the left side. The bottom part shows the values of the hyperparameters. The numeric hyperparameter is visualized with a circle where the ratio of the circle is proportional to the hyperparameter value. This helps us identify if a hyperparameter was changed. The selection strategy kinds are depicted with a triangle. We assign a color to each selection strategy, the figure legend shows the supported strategies and their associated colors.

We observe that the main goal considered for all participants was the increasing of the test coverage. Figure 22 shows the components observed, and the hyperparameter values modified by each participant for the test generations. 10 of the 14 participants (P2, P12, P14-P16, P18-P22) observed the middle panel at least 50% of the time after executing the tool with a given configuration. In similar way, it happened with the Coverage Evolution panel, which was looked at by nine participants (P2, P3, P5, P11, P16, P18-P19, P21-P22) at least 50% of the times after test generations were made. The principal reason because these two components were more considered, in comparison with the rest, is because they contain the coverage information achieved through the generations, also they show how the evolution goes, i.e. if there is an increasing of the coverage in the generation or not, the covered methods or branches of the target class, which tests were selected to survive until the last generation, etc. Another important component for nine participants (P3, P5, P12, P14, P16-P20) is the Similarity panel, the participants modified the hyperparameter values not just to get higher coverage, but also a greater diversity between tests. And finally, four participants (P15, P17, P18, P21) paid attention also in the Contribution panel, because they took in consideration the method and class coverage increasing for tuning the values.

Participants change different hyperparameters during task T3. Five participants (P15, P16, P3, P5, P12) reduced the number of generations in some executions because they saw that after a certain number the coverage did not increase anymore. Three participants (P16, P5, P21) increased the generation number because they observed that there was a gradual increase in the coverage and they wanted to see if the coverage would still increase. While eight participants (P2, P5, P11, P15, P17, P19, P20) considered, besides the coverage, also the gradual evolution. It means, they took into account the new methods or branches covered in the next generations after the first. And four participants modified the mutation rate in order to diversify the tests.

Conclusions. Based on our observations, to complete the task related to hyperparameter tuning, participants had the following behaviors:

– 10 of the 14 participants observed the evolution panel (middle panel) at least 50% of the time after executing the tool with a given configuration to analyze the tests that increment the coverage regarding their parents;
– 9 of the 14 participant observed the evolution panel to analyze the coverage variations along generations;
– 9 of the 14 participant observed the similarity panel to get an overview of the test diversity;
– The two most changed hyperparameters are the population size and the number of generations.

6.3.5 Discussion & user feedback

A number of items are worth discussing.

Customization. P1 suggested that some components of TestEvoViz could be optional for regular users of visualization, this participant said that the

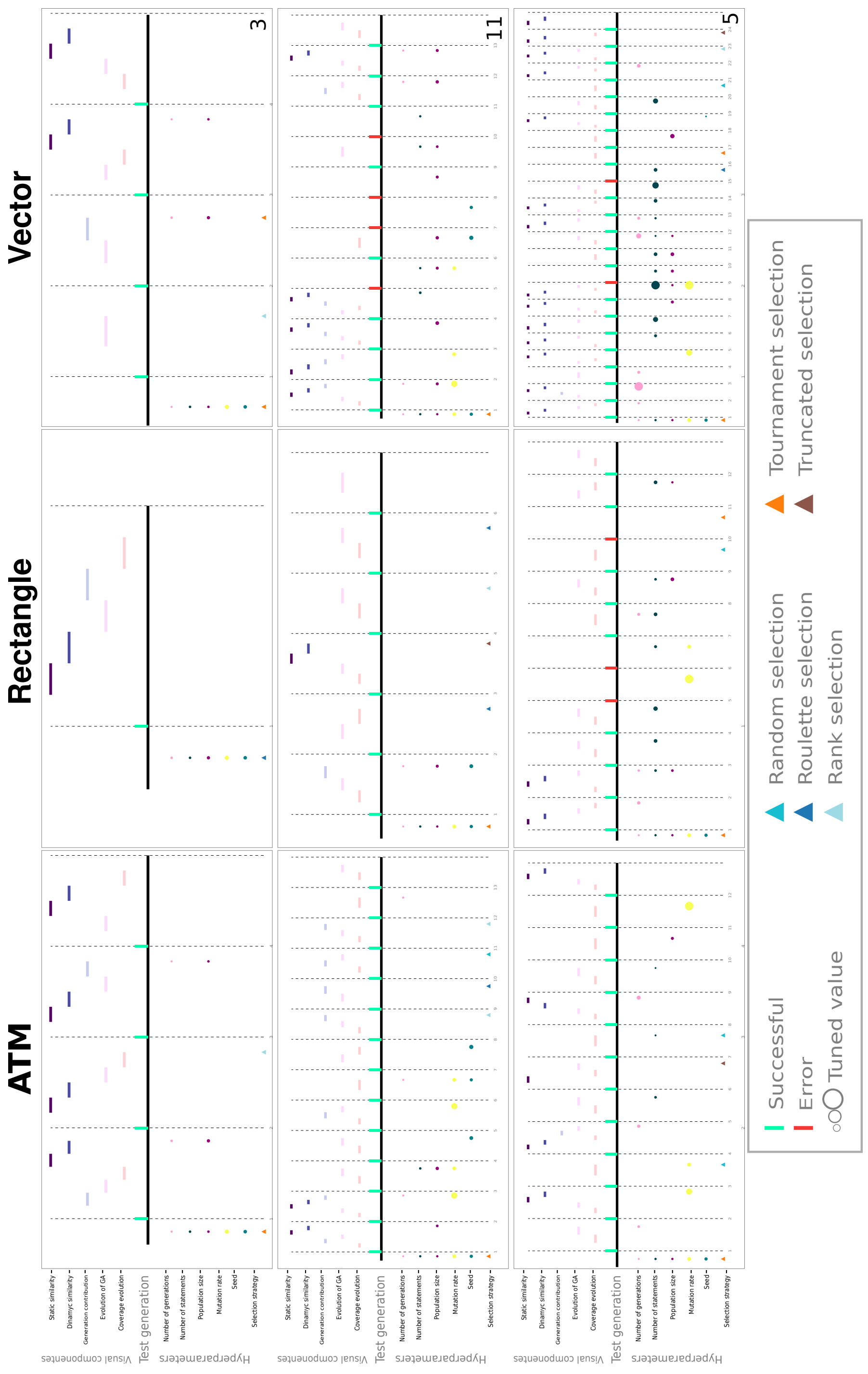


Fig. 22: Session visual summary – Summary of the session of three participants during Task T3. The visual summary of the remaining participants may be found in the appendix.

center panel showing the test case evolution is the most important, and the rest can be activated on demand. A similar suggestion is of P22, who said that the coverage value achieved is very important, and if any user would like to see details about the evolution (the similarity, methods or branches that were covered), the other panels could be activated. On the other hand, P3 suggested that very similar tests could be visually grouped in a box, and a way to see the differences between tests of the same generation would be helpful. While P19, P21, and P22 proposed the tool be capable of showing code differences between parents and children. Other suggestions were given by P7, P3, and P22, who proposed highlighting tests of the similarity chart when a user is interacting with a test of the center panel and vice versa. P13, P14, P18, and P20 suggested incorporating descriptions of the components of the tool, in order to have quick access to the information in case of forgetfulness.

Discarded tests. To reduce the width of the visualization and the amount of information, TestEvoViz does not show generated tests that do not contribute to the final generation. However, P6 and P17 were curious to understand which test cases were discarded to see if these tests were responsible for the increment of the class coverage.

Branch granularity analysis. The test similarity metrics consider similarity at method level and not at branch level, however, two participants (P3 and P9) wanted to contrast the branches that were executing two tests to understand the exact difference in a number of cases. That comparison can be possible through inner boxes' popups. But like the popup, it is just visible with the interaction, and this can complicate the comparison a bit.

Similarity. Seven participants (P2, P7, P11-P13, P16-P17) expressed that the similarity panel was hard to understand when the population size was bigger. Initially, we designed the similarity panel to provide an overview of how similar the generated tests are. However, a detailed exploration is not possible without many complex interactions with the visualization. Therefore, we conclude that dedicated tools to detailed similarity comparison are needed.

Hyperparameter tuning. In order to modify the hyperparameter values and define the final value, P2, P15, and P19 suggested a summary table that shows the values of the hyperparameters, and the coverage achieved using those values. P22 said that it would be helpful to have two windows, one showing the graphic, produced by the visualization tool, of a previous execution, and the other the graphic of the current execution.

7 Threats to Validity

As with any empirical evaluation, our user study has a number of threats to validity. The following paragraphs report a number of them.

Scalability. TestEvoViz uses a grid layout, which makes the overall visualization size depend on the population size and the number of generations.

Therefore, a larger visualization typically requires scrollbars, which may involve more interaction from a practitioner to enjoy the visualization. To mitigate the negative 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 a method. In the 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*[6]. SmallSuiteGenerator implements the original algorithm of EvoSuite presented at [11]. 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 SmallSuiteGenerator uses to generate tests.

Whole test suite generation approach. Our visualization targets the whole suite test generation approach implemented by EvoSuite, which considers one target at the time and a single fitness function. However, there is another evolutionary algorithm that uses a many-objective optimization algorithm. TestEvoViz may need to adapt a number of their components to assess the evolution process of different test generation techniques [4].

Participants & session load. It is difficult to find people with an expertise in genetic algorithm and/or test generation. Mainly because test generation tools are not yet widely used in the industry. Participants without a background in the area have more difficulties using our visualization, and this is one of the reasons that the sessions were longer. Although we send a open invitation to participate in our study, we also personally invite people with knowledge of test generation to reduce this threat. Therefore, we believe that our study capture the feedback of a great variety of potential end users.

Project under study. The projects used in tasks `T1` and `T2` were not developed by the participants, and they were unfamiliar with the tested code. However, this is not an issue since in practice testers often test code developed by others. We choose projects whose domain is simple enough to be understood and tested in a reasonable time. For task `T3`, similar to previous studies [22], we select classes that are well know for all participants, this is important since

[6] `https://github.com/OBJECTSEMANTICS/SmallSuiteGenerator`

they actually needed to analyze the resulting generated test to tune the hyper-parameters. We also take into account the time needed for the tool to generate tests, since for task T3 participants need to generated test multiple times. Nevertheless, the selected projects and classes for the study limit the external validity of our study.

Conclusion. We manually analyze and categorize participants' answers, and actions while they were using the visualization. Therefore, the conclusions and discussions presented in the paper are limited by our perspective.

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.

8 Related Work

Though genetic algorithms were proposed in the 60s, 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 fitness along each individual of a generation is usually represented using charts 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 circles [6] to highlight coverage variations between iterations. Previous studies used Cartesian layouts to visualize dynamic graphs, but normally applied to software evolution and call graphs [23, 24, 25, 26]. We use a Cartesian layout to relate generated tests with their corresponding iterations.

We associated a number of metrics to each node inspired in polymetric view [27, 28, 29]. Polymetric views are commonly used to visually map entity metrics in a glyph box glyph, this technique have been used to enhance nodes within a graph. For instance, call graphs, and dependency graphs. At difference of previous works, we use polymetric views to visualize different properties of a given generated test along the evolution. Relationships between nodes with their ancestors are represented as edges [30, 31]. Edge lines were inspired from hierarchical bundle edges [32]. Similar to previous work, bezier lines help us to void dense edge collision and facilitates the analysis.

Hart et al. [31] 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). Our approach use similar structure to show the ancestors, however, our approach show all ancestors of the final generation and provide highlights the ancestors of a particular node when clicking it.

Romero et al. [33] use color maps to visualize the individuals and chromosomes of the population. It use a matrix layout were each column is a generation, the cell of the matrix contains the fitness value of each element. Ito et al. [34] 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". In contrast, we use a graph to represent the relationship between elements.

Farooq et al. [35, 36] 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 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 for all generations.

Tomida et al. [37] propose a technique to visualize the evolution process of automated program repair. It is based on 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. At difference of this work, we focus in test generation rather than program repair. The nodes within our graph highlight test related metrics instead the algorithm operations.

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.

Regarding the evaluation, all previous approaches present a number of examples and case studies to highlight the usefulness of their proposed visualization [35, 36, 37, 34, 33, 31]. For instance, applying the visualization to understand how the genetic algorithm reaches a number of solutions for traditional problems like the rastrigin problem [33], a timetabling problem, a jobshop scheduling problem, and Goldberg and Horn's long-path problem [31]. In this paper, we present a dedicated user study with 22 participants, and analyze the application of the genetic algorithm in a different domain which is test generation.

9 Conclusion and Future Work

TestEvoViz is an interactive visualization approach that help developers to introspect a genetic algorithm-based test generation process. It depicts different concepts and decisions made by the genetic algorithm through various related visual components. We illustrated the applicability of our proposed visualization thought two real world case studies. Complementary, we also performed an user study involving 22 participants that use TestEvoViz to analyze, compare and tune the test generation processes. Our finding shows that participants mainly focus on the code coverage and test diversity as principal attributes of the generated test. As a consequence, the most used visual components by our participants are: (i) the similarity panel, which brings an overview of the similarity between generated test; (ii) the evolution panel, which depicts how the different test were evolving across generations, and (iii) the coverage evolution panel which gives the minimum, maximum and average coverage for each generation.

We believe TestEvoViz extends the State of the Art in comprehending evolution-based test generation by means of an expressive, intuitive, and effective visualization. However, TestEvoViz may be considered as a contribution on which numerous extensions can be built upon. In particular:

- Assertions are currently not represented in our visualization. Our future work contemplates visualizing assertions as a combination of the program coverage by the assertions and the syntactic components composing that assertion;
- From an initial configuration of hyperparameters, TestEvoViz visualizes the evolution of generated unit tests. As we have shown, assessing the impact of some changes in the initial configuration is a manual task that requires spotting differences between multiple visualizations. As a future work, we will design a new visualization that shows the difference of the evolution between two or more different initial configurations. Some ingredient from our previous work will be considered [30];
- TestEvoViz has been designed to accommodate with the execution model of EvoSuite. However, nothing prevents our visualization to operate with a different execution model and metaheurstics. For example, our visualization may be used to support other optimization techniques, such as Hill Climber, or reinforcement learning [38].

Acknowledgment

Juan Pablo Sandoval Alcocer thanks ANID FONDECYT Iniciación Folio 11220885 for supporting this article. Alexandre Bergel is grateful to the FONDECYT Regular project 1200067 for having partially sponsored the work presented in this article. We are deeply grateful to Lam Research (4800054170 and 4800043946). We thank Renato Cerro for his help reviewing an early draft of the manuscript.

Declarations

Funding and/or Conflicts of interests/Competing interests

Juan Pablo Sandoval Alcocer thanks ANID FONDECYT Iniciación Folio 11220885 for supporting this article. Alexandre Bergel is grateful to the FONDECYT Regular project 1200067 for having partially sponsored the work presented in this article.

Appendix

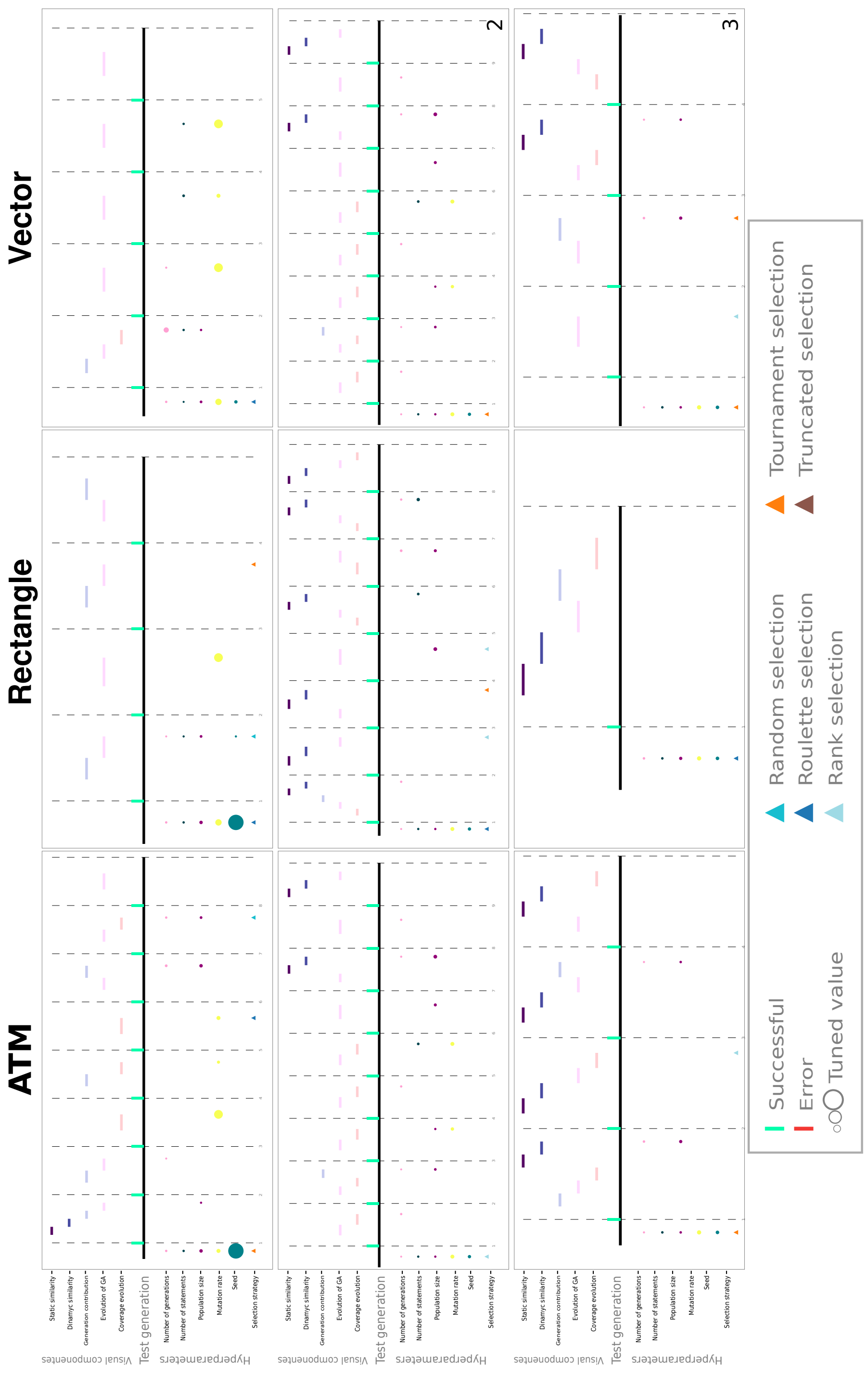


Fig. 23: Session visual summary of participants during Task T3

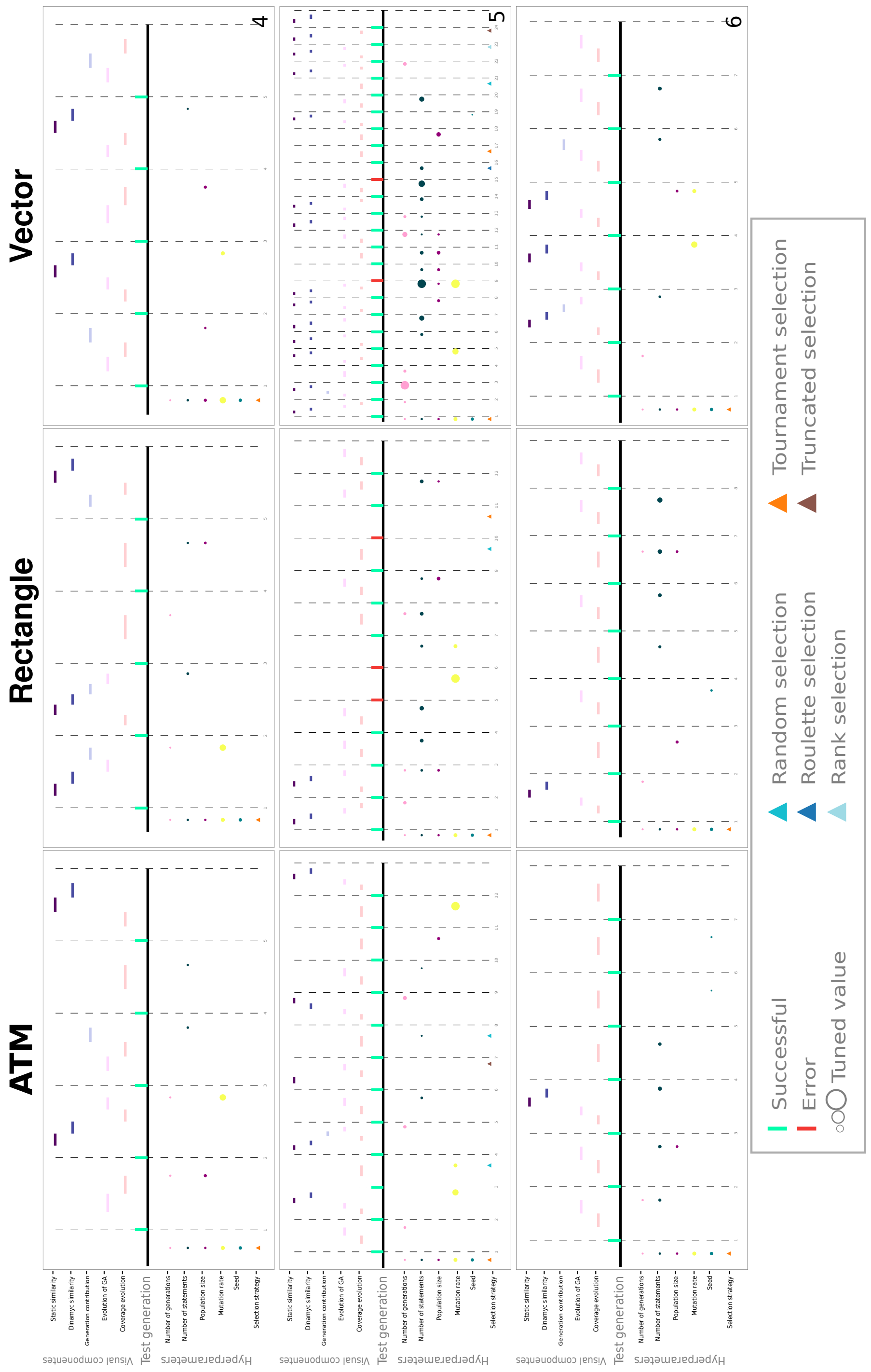


Fig. 24: Session visual summary of participants during Task T3

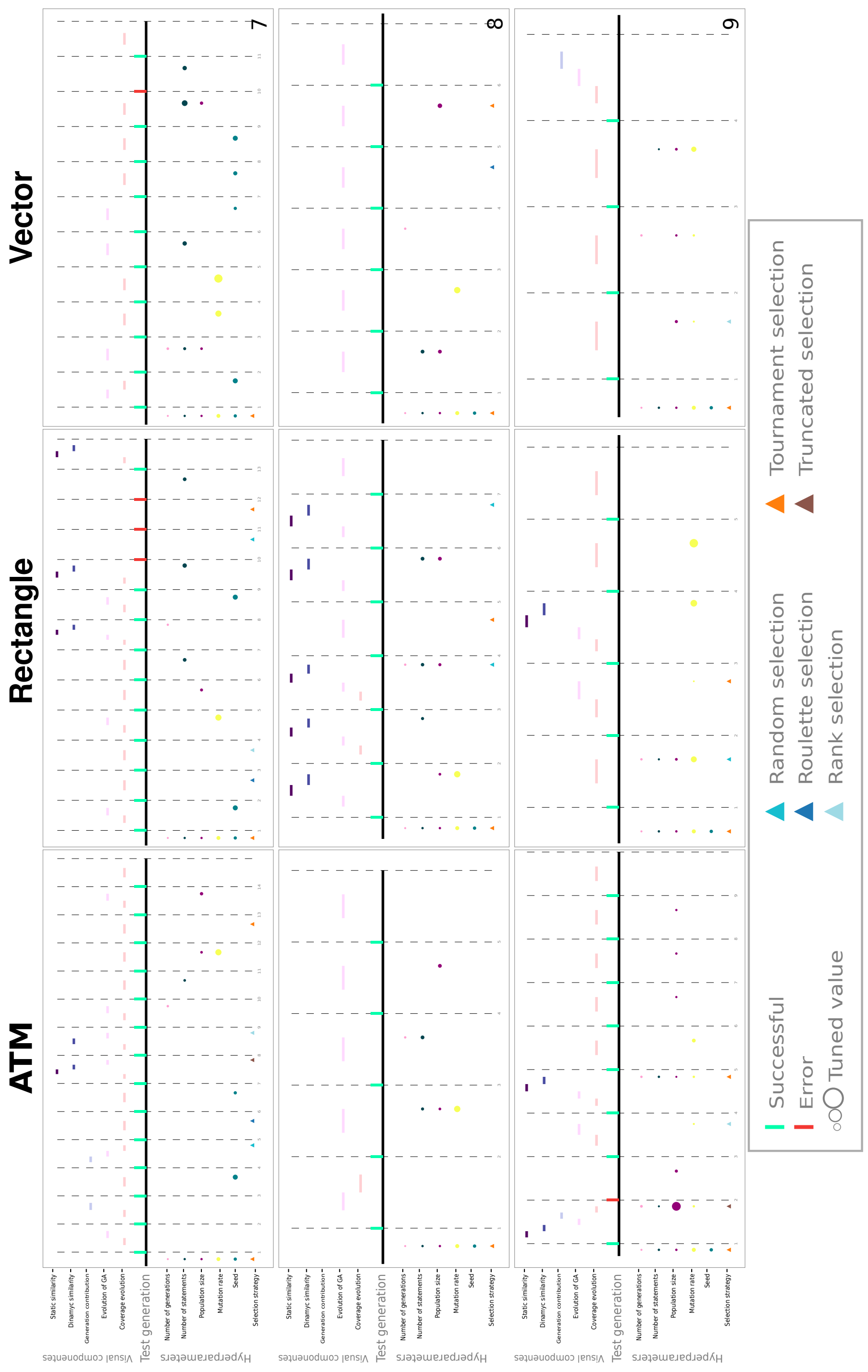


Fig. 25: Session visual summary of participants during Task T3

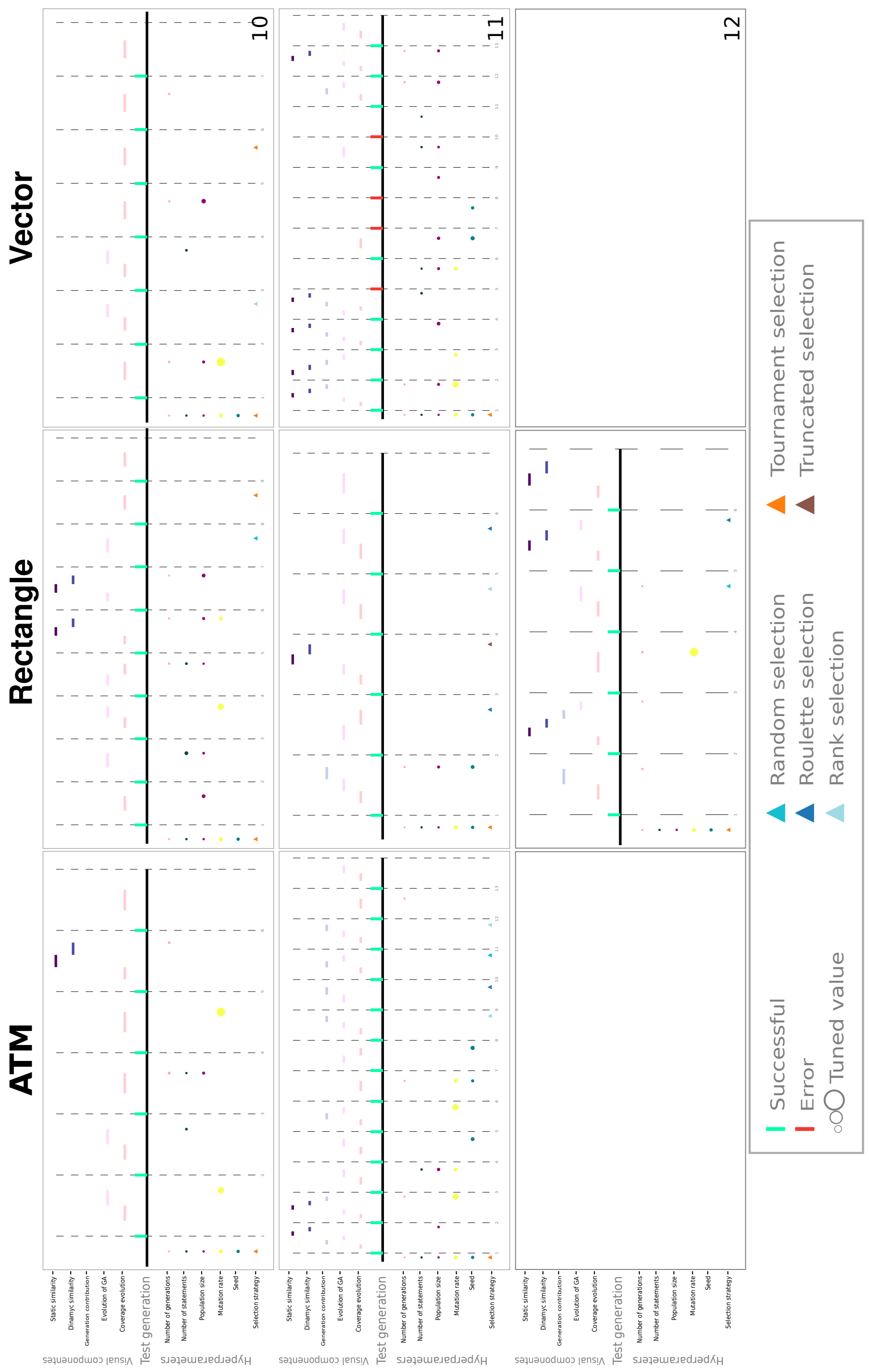


Fig. 26: Session visual summary of participants during Task T3

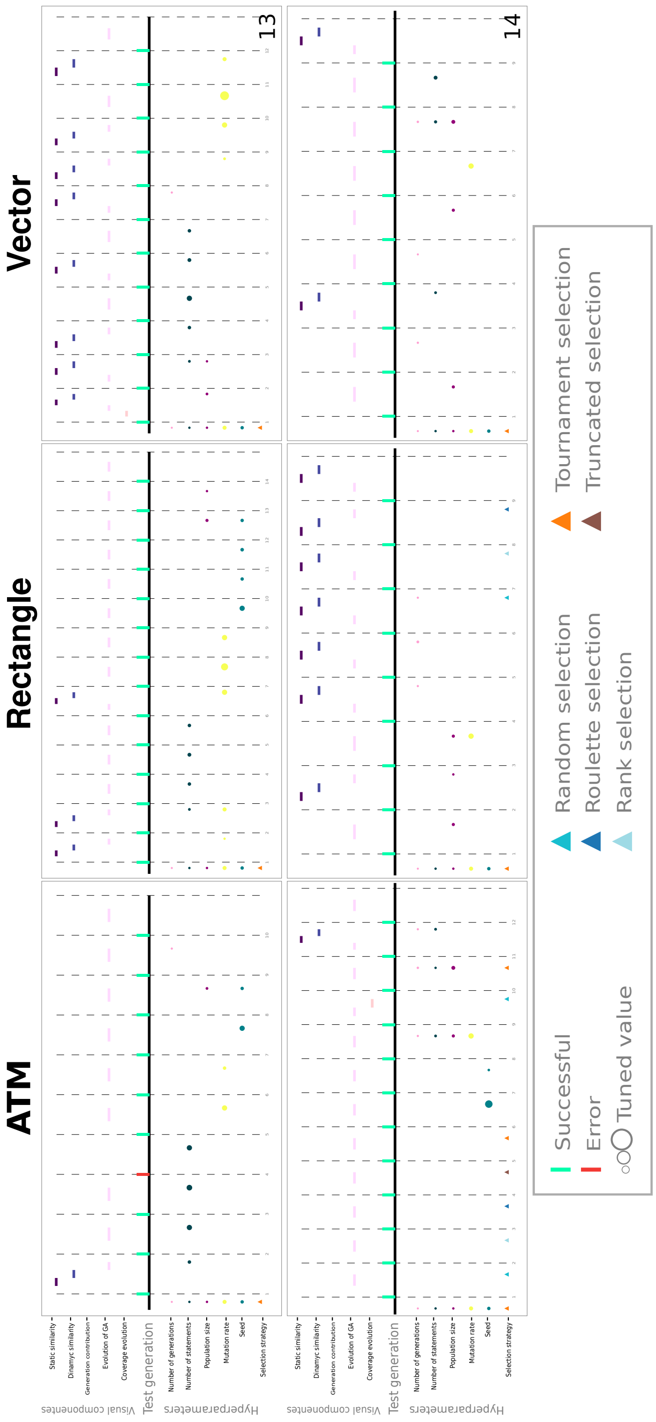


Fig. 27: Session visual summary of participants during Task T3

References

1. Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. The fuzzing book. In *The Fuzzing Book*. Saarland University, 2019. Retrieved 2019-09-09 16:42:54+02:00.
2. Benjamin Danglot, Oscar Vera-Perez, Zhongxing Yu, Andy Zaidman, Martin Monperrus, and Benoit Baudry. A snowballing literature study on test amplification. *Journal of Systems and Software*, 157:110398, 2019.
3. Gordon Fraser and Andrea Arcuri. Evolutionary generation of whole test suites. In *International Conference On Quality Software (QSIC)*, pages 31–40, Los Alamitos, CA, USA, 2011. IEEE Computer Society.
4. A. Panichella, F. M. Kifetew, and P. Tonella. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering*, 44(2):122–158, 2018.
5. José Campos, Yan Ge, Nasser Albunian, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. An empirical evaluation of evolutionary algorithms for unit test suite generation. *Information and Software Technology*, 104:207–235, 2018.
6. 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 (VISSOFT)*, pages 28–32, 2019.
7. A. Cota Vidaure, E. Cusi Lopez, J. P. Sandoval Alcocer, and A. Bergel. TestEvoViz: Visual introspection for genetically-based test coverage evolution. In *2020 Working Conference on Software Visualization (VISSOFT)*, pages 1–11, 2020.
8. Gordon Fraser and Andreas Zeller. Mutation-driven generation of unit tests and oracles. In *Proceedings of the ACM International Symposium on Software Testing and Analysis*, ISSTA '10, pages 147–158. ACM, 2010.
9. Andrea Arcuri and Gordon Fraser. On parameter tuning in search based software engineering. In *Proceedings of the Third International Conference on Search Based Software Engineering*, SSBSE'11, pages 33–47. Springer-Verlag, 2011.
10. Carlos Pacheco and Michael D. Ernst. Randoop: Feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*, OOPSLA '07, page 815–816, New York, NY, USA, 2007. Association for Computing Machinery.
11. Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276 –291, Feb. 2013.
12. Andrea Arcuri and Gordon Fraser. On the effectiveness of whole test suite generation. In *Proceedings of the Sixth International Conference on Search Based Software Engineering*, SSBSE'14, pages 1–15, Berlin, Heidelberg, 2014. Springer-Verlag.
13. S Shamshiri, JM Rojas, L Gazzola, G Fraser, P McMinn, L Mariani, and A Arcuri. Random or evolutionary search for object-oriented test suite generation?, Mar 2018.
14. A. Panichella, S. Panichella, G. Fraser, A. A. Sawant, and V. J. Hellendoorn. Revisiting test smells in automatically generated tests: Limitations, pitfalls, and opportunities. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 523–533, 2020.
15. Fabian Beck, Michael Burch, Stephan Diehl, and Daniel Weiskopf. A taxonomy and survey of dynamic graph visualization. *Comput. Graph. Forum*, 36(1):133–159, jan 2017.
16. Gordon Fraser and Franz Wotawa. Increasing diversity in coverage test suites using model checking. In *Proceedings of the 2009 Ninth International Conference on Quality Software*, QSIC '09, page 211–218, USA, 2009. IEEE Computer Society.
17. Nadia Alshahwan and Mark Harman. Augmenting test suites effectiveness by increasing output diversity. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, page 1345–1348. IEEE Press, 2012.
18. Sugeerth Murugesan, Kristofer Bouchard, Jesse Brown, Mariam Kiran, Dan Lurie, Bernd Hamann, and Gunther H Weber. State-based network similarity visualization. *Information Visualization*, 19(2):96–113, 2020.

19. Juan Pablo Sandoval Alcocer, Harold Camacho Jaimes, Diego Costa, Alexandre Bergel, and Fabian Beck. Enhancing commit graphs with visual runtime clues. In *2019 Working Conference on Software Visualization (VISSOFT)*, pages 28–32, 2019.
20. Francisco Lopez Luro and Veronica Sundstedt. A comparative study of eye tracking and hand controller for aiming tasks in virtual reality. In *Proceedings of the 11th ACM Symposium on Eye Tracking Research & Applications*, ETRA '19, New York, NY, USA, 2019. Association for Computing Machinery.
21. John Brooke. SUS: A quick and dirty usability scale. *Usability Evaluation in Industry*, 189, 1996.
22. Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. Does automated white-box test generation really help software testers? In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, page 291–301, New York, NY, USA, 2013. Association for Computing Machinery.
23. Michele Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. *International Workshop on Principles of Software Evolution (IWPSE)*, 09 2001.
24. Fabian Beck, Michael Burch, Corinna Vehlow, Stephan Diehl, and D. Weiskopf. Rapid serial visual presentation in dynamic graph visualization. In *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC*, pages 185–192, 09 2012.
25. Juan Pablo Sandoval Alcocer, Alexandre Bergel, Stéphane Ducasse, and Marcus Denker. Performance Evolution Blueprint: Understanding the impact of software evolution on performance. In *Proceedings of the 1st IEEE Working Conference on Software Visualization*, VISSOFT, pages 1–9. IEEE, 2013.
26. Juan Pablo Sandoval Alcocer, Fabian Beck, and Alexandre Bergel. Performance Evolution Matrix: Visualizing performance variations along software versions. In *Proceedings of the 7th IEEE Working Conference on Software Visualization*, VISSOFT, 2019.
27. M. Lanza and S. Ducasse. Polymetric views - a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, 2003.
28. Alexandre Bergel, Felipe Bañados, Romain Robbes, and Walter Binder. Execution profiling blueprints. *Software: Practice and Experience*, 42, 09 2012.
29. Alexandre Bergel and Vanessa Peña. Increasing test coverage with hapao. *Science of Computer Programming*, 79:86 – 100, 2014. Experimental Software and Toolkits (EST 4): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT-3 2010).
30. 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)*, pages 1–9, Sep. 2013.
31. E. Hart and P. Ross. Gavel - a new tool for genetic algorithm visualization. *IEEE Transactions on Evolutionary Computation*, 5(4):335–348, 2001.
32. D. Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):741–748, 2006.
33. Gustavo Romero, Juan J. Merelo Guervós, Pedro A. Castillo Valdivieso, J. G. Castellano, and Maribel García Arenas. Genetic algorithm visualization using self-organizing maps. In *Proceedings of the 7th International Conference on Parallel Problem Solving from Nature*, PPSN VII, page 442–451, Berlin, Heidelberg, 2002. Springer-Verlag.
34. Shin-Ichi Ito, Yasue Mitsukura, Hiroko Nakamura Miyamura, Takafumi Saito, and Minoru Fukumi. *A Visualization of Genetic Algorithm Using the Pseudo-Color*, page 444–452. Springer-Verlag, Berlin, Heidelberg, 2008.
35. Humera Farooq, Nordin Zakaria, and Muhammad Tariq Siddique. An interactive visualization of genetic algorithm on 2-d graph. *Int. J. Softw. Sci. Comput. Intell.*, 4(1):34–54, January 2012.
36. Humera Farooq and Muhummad Tariq Siddique. A comparative study on user interfaces of interactive genetic algorithm. *Procedia Computer Science*, 32:45 – 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).

37. 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)*, pages 23–27, 2019.
38. Afonso Fontes, Gregory Gay, Francisco Gomes de Oliveira Neto, and Robert Feldt. Automated support for unit test generation: A tutorial book chapter. *CoRR*, abs/2110.13575, 2021.

Andreina Cota Vidaurre is a machine learning engineer at Centro Nacional de Inteligencia Artificial (CENIA), Chile. Her research interests include software maintenance and evolution, software testing, software visualization, artificial intelligence, machine learning, and data analysis.

Evelyn Cusi López is a Data Scientist (Research and Development) at Project44, Germany. She received her BSc (Hons) in System Engineering from Mayor de San Simón University, Bolivia. Evelyn worked as software engineer for three years and she was part of the GSoC program (Google Summer of Code) as a contributor in 2019 and as a mentor in 2021 for the Pharo consortium. She is interested in studying and improving software quality and contributing to open source. Her research interests include software maintenance and evolution, artificial intelligence, machine learning, data analysis and data visualization.

Juan Pablo Sandoval Alcocer is an Assistant Professor at the Department of Computer Science, School of Engineering, Pontificia Universidad Católica de Chile. He received the Ph.D. degree in computer science from University of Chile, Chile, in 2016. He is part of the Software Engineering and Intelligent Systems Laboratory (SEIS Lab). His research interests lie in software engineering, more specifically in the fields of software maintenance, mining software repositories, software performance, software visualization, and search-based software testing. He participated as a reviewer expert in various prestigious conferences and journals in the field including: EMSE, IEEE VIS, VISSOFT, JSS, and IST. He is also member of the Pharo community.

Alexandre Bergel is computer scientist at RelationalAI, Switzerland. Until 2022, he was Associate Professor and researcher at the University of Chile. Alexandre Bergel and his collaborators carry out research in software engineering. His focus is on designing tools and methodologies to improve the overall performance and internal quality of software systems and databases by employing profiling, visualization, and artificial intelligence techniques. Alexandre Bergel is member of the editorial board of Empirical Software Engineering and authored four books in the fields of data visualization, intelligence artificial, and the Pharo programming language.