A. Ramírez, P. Delgado-Pérez, K. J. Valle-Gómez, I. Medina-Bulo and J. R. Romero, "Interactivity in the Generation of Test Cases with Evolutionary Computation," *2021 IEEE Congress on Evolutionary Computation (CEC)*, 2021, pp. 2395-2402, doi: 10.1109/CEC45853.2021.9504786.

Version: Accepted Version

© 2021 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Interactivity in the Generation of Test Cases with Evolutionary Computation

Aurora Ramírez
Dpto. Informática y Análisis Numérico
University of Córdoba
Córdoba, Spain
aramirez@uco.es

Pedro Delgado-Pérez
Dpto. de Ingeniería Informática
University of Cádiz
Cádiz, Spain
pedro.delgado@uca.es

Kevin J. Valle-Gómez
Dpto. de Ingeniería Informática
University of Cádiz
Cádiz, Spain
kevin.valle@uca.es

Inmaculada Medina-Bulo
Dpto. de Ingeniería Informática
University of Cádiz
Cádiz, Spain
inmaculada.medina@uca.es

José Raúl Romero
Dpto. Informática y Análisis Numérico, University of Córdoba
Andalusian Research Institute in Data Science
and Computacional Intelligence (DaSCI),
Córdoba, Spain
jrromero@uco.es

***Abstract*—Test generation is a costly but necessary testing activity to increase the quality of software projects. Automated testing tools based on evolutionary computation principles constitute an appealing modern approach to support testing tasks. However, these tools still find difficulties to detect certain types of plausible faults in real-world projects. Besides, recent studies have shown that, in general, automatically-generated tests do not resemble those manually written and, consequently, testers are reluctant to adopt them. We observe two key issues, namely the opacity of the process and the lack of cooperation with the tester, currently hampering the acceptance of automated results. Based on these findings, we explore in this paper how the interaction between current tools and expert testers would help address the test case generation problem. More specifically, we identify a number of interaction opportunities related to the object-oriented test case design driven to boost their readability and detection power. Using EvoSuite as base implementation, we present a proof of concept focused on the possibility to integrate readability assessment of the most promising test suites into a genetic algorithm.**

***Index Terms*—software testing, interactive search-based software engineering, test generation, mutation testing**

I. Introduction

Testing is a critical phase in the development of software projects to improve their quality. This phase however is known to be costly, taking up a great percentage of total project resources [1]. With the increasing complexity of current software developments in the industry, manual testing becomes unapproachable or simply leads to poorly-designed test suites. To face this problem, a large body of research has sought to replace manual steps with automated and efficient techniques that guide the generation of effective test suites [2]. As a result, many proposals have appeared in the field of *automated test generation*, with an emphasis on the application of evolutionary approaches, especially for test data generation [3].

Unlike test data generation, test case and test suite generation imply the design of test scenarios. In the case of object-oriented software, a test scenario for a class consists of a sequence of method calls, followed by an evaluation of the results (i.e. outputs and state of objects) [4]. Apart from supplying data that is sensitive enough to reveal deviations from the expected functionality, object-oriented test case generation brings some other challenges, such as setting a convenient state for the objects, sequencing invocations in an appropriate order or adding proper assertions that often involve examining different variables of the program. A remarkable contribution in the last decade is EvoSuite [5], a tool based on evolutionary computation that generates test cases, independently or as part of whole test suites[1], able to detect different types of faults seeded with mutation testing, a well-known technique to assess and improve the quality of test suites.

Recent papers have evaluated the current state of test generation tools [6], [7] including EvoSuite and Randoop [8]. Despite the great advances, these studies reveal some limitations in the application of these automated tools that prevent most practitioners from embracing them in their daily practice [9], [10]. Among others, two relevant obstacles that remain unresolved are the fault-detection effectiveness [7] and the skepticism of testers regarding the readability of the resulting tests [11]. These findings suggest that neither completely manual nor fully automated test generation is the best option. In this sense, our idea is to combine automatic and manual testing, i.e. the efficiency of search-techniques and the know-how of human testers, to address these issues.

Some authors have already pointed to the benefits of in-

Work partially funded by the European Commission (ERDF), the Spanish Ministry of Science, Innovation and Universities [RTI2018-093608-BC33 and RED2018-102472-T], the University of Córdoba (Plan Propio - mod. 2.4), and the University of Málaga.

978-1-7281-8393-0/21/$31.00 ©2021 IEEE

[1]From now on in this paper, we use the term *test generation* to refer to *test case* or *test suite generation* indistinctly.

corporating human knowledge into the test generation process via interactive optimization [10], [12]. Interactive approaches encourage the active participation of engineers, and have been successfully applied in several areas of search-based software engineering (SBSE) [13]. Focusing on search-based software testing (SBST), Marculescu et al. proposed an interactive approach to allow domain-specialists to take part in the test data generation process [14], [15]. Evaluated in the context of an industrial collaboration, their system represents a satisfactory example of the possibilities that interactivity brings to SBST.

Inspired by these works, this paper seeks to make more visible the role that interactivity could play as part of search-based test generation. First, we analyze those characteristics of object-oriented testing that make it difficult to be fully automated. Then, we study how interactive optimization could solve some of the identified challenges. Our hypothesis is that, by putting the tester "in the loop", an SBST tool would be able to take advantage of their skills and knowledge of the system under test (SUT) to design meaningful tests. Such a tester-centered interactivity could augment current capabilities of test generation, enabling them to find out reasons behind fault detection deadlocks and make tests more readable. Lastly, we present a proof-of-concept in which some of the interactive options are implemented in EvoSuite with the aim of incorporating readability assessments of the most promising test suites. We incorporate readability assessment at scheduled interaction moments to favor those those test suites which are more aligned with the tester's preferences, and we show an execution of the resulting interactive process. We expect that the proposed interaction opportunities will contribute to overcome current tool limitations while increasing the tester acceptance on automated results, issues that become especially relevant in industrial settings.

The paper is structured as follows. Section II introduces concepts surrounding SBST and interactive SBSE. Related work is collected in Section III. Section IV discusses current limitations and identifies the requisites that guide our proposal. The interactive opportunities for test generation are described in Section V. Section VI presents the proof of concept. Finally, Section VII outlines conclusions and future work.

II. Background

A. Search-based software testing

Automated test generation can be defined as the process in which an automatic entity is applied to generate new test cases for testing the adequacy of existing software [16]. However, since exhaustive testing is normally impractical, cost-effective solutions are required to address this problem and reach fault-revealing test cases. In the context of object-oriented systems, SBST has revealed as a powerful approach to perform this activity at the class level. Test generation can be expressed as a search problem, where candidate solutions represent the sequences of method calls and their parameters. The search strategy is guided by means of a fitness function, based on satisfying testing goals and usually expressed as an aggregation of coverage functions. The search-based EvoSuite tool exemplifies its success in evolving existing test suites into new ones with a higher fault-detection capability [5].

Among other approaches, EvoSuite applies a genetic algorithm (GA) to evolve a population of test cases or whole test suites (individuals) for Java classes, mainly represented as a sequence of method and constructor calls with values for their parameters. Thanks to selection and variation operators (mutation and crossover), these individuals are iteratively modified in subsequent generations to approach an optimization target expressed in the fitness function: maximize one or more testing criteria [17]. Among other testing criteria, EvoSuite can apply a mutant-driven approach, that is, the GA is guided towards the detection of the largest possible number of seeded artificial faults based on common programming mistakes (called *mutants*). Secondary objectives can also be defined to consider some other additional goals, such as decreasing the size of the test suite. These secondary objectives are only activated to break ties when comparing candidate solutions. The test generation process is completed with the inclusion of assertions based on mutation analysis (when the outputs of the mutant and the original program differ, we say that the mutant is detected or killed), and with the minimization of the test sequences to improve their readability in general.

B. Interactivity in SBSE

Interactive SBSE (iSBSE) has been formalized as an emergent subarea within SBSE in a literature review [13], which collects the interaction mechanisms and applications proposed so far. iSBSE promotes the active participation of software engineers by providing intermediate results for their inspection. Their feedback is later integrated into either the problem formulation or the search process so that the algorithm progressively adapts the search to the human's preferences [18]. Interactivity is expected to alleviate some of the limitations of SBSE techniques [19], such as problem oversimplification, the inability of fitness functions to capture subjective criteria, or the lack of trust on automated results [18]. Nevertheless, iSBSE also poses new challenges like dealing with the inherent fatigue or the information overload [18], [19]. Several decision factors influence the design of iSBSE proposals, which should be properly adapted to the domain [13]:

1) *Type of interactive algorithm*, which mostly depends on the goal(s) pursued with the interaction. Non-exclusive categories here are interactive reoptimization, aimed at refining the problem definition during the search; human-based evaluation, in which the human takes part in the solution assessment; preference-based interactivity, focused on the progressive incorporation of human preferences; and human-guided search, allowing the human to directly select or modify solutions.
2) *Type of feedback*, which determines the available user actions, usually solution evaluation, selection, comparison or modification. For evaluation, the human can be asked to assign a fitness value, adjust weights for the objectives, rank solutions, provide a score by choosing from preset values or reward/penalize characteristics.

3) *Interaction schedule*, which encompasses the moment and frequency of interaction, and the selection of solutions to be shown, including the number of solutions, the criteria to choose them from the population and the level of solution detail (complete or partial).
4) *Information integration*, which requires deciding about how long the feedback influences the search, e.g. during some generations or the whole process, and whether it can be reverted or modified in next interactions.

III. Related work

Important research efforts have been devoted to design SBST methods for finding faults in object-oriented code [16]. Nevertheless, several authors have pointed out that SBST algorithms still present some limitations, specially when it comes to detect bugs in real-world software systems [10], [17], [20]. Rojas et al. focused on the problem of using code coverage to guide evolutionary algorithms [10]. In their opinion, the fact that test assertions are not usually included in the optimization process might be a cause of low performance, and therefore other objectives are needed to produce effective tests. Recently, these authors carried out an experiment to evidence that state-of-the-art tools indeed fail to find real faults [21]. A similar study concludes that relying on coverage criteria only, even if multiple measures are combined, is insufficient to detect some real faults with EvoSuite [17]. Finding appropriate fitness functions is essential not only to improve fault detection rates, but also to achieve human-competitive results. For instance, Gay [22] proposes defining SUT-specific functions based on the system context to better simulate how humans develop test cases. Other authors agree that additional testing design factors beyond structural coverage should be considered in the context of industrial safety-critical control software [20], an scenario where manual testing has shown better detection capabilities for certain types of faults.

Other empirical experiments comparing manual tests and those automatically generated have been carried out with the aim of evaluating the acceptance of SBST results. In this sense, the time needed to understand the intention of non-manual test cases is usually greater due to the lack of comments and descriptive names in the code [11]. This study also points out that developers are skeptical of failures reported by automated tests, and perceive them as less helpful than those manually written for maintenance tasks. The cost-benefit associated to checking SBST results has been analyzed in an industrial context too [20]. However, improving the readability of test cases still represents an open challenge because of its subjective nature [10]. In an attempt to measure readability, Daka et al. trained a regression model to infer desirable code properties, such as identifier length or the use of code constructs (loops, branches, assertions, etc), from a collection of human-rated test cases. Their readability prediction model was used in a postprocessing step to produce more readable tests cases while keeping their original coverage [23], and also as an additional objective in a multi-objective approach [24]. A controlled experiment revealed that more readable tests help testers make decisions faster, though the selection of variable names still need improvements to become more flexible.

Different authors have mentioned the potential of interactive approaches to address other software testing tasks, such as providing test oracles [10]. Marculescu et al. were the first to explore interactive evolutionary computation for software testing activities [25]. They emphasized the importance of expert knowledge and SUT context to guide the search. Since their interaction is focused on solution evaluation, they proposed choosing a small number among the best solutions to avoid fatigue. Their following works delve further into the idea of "user as fitness function" for black-box testing in the context of an industrial embedded software. More specifically, the domain specialist is asked to periodically re-assign weights to the 11 objectives of a differential evolution algorithm for test data generation [14]. A visualization module allows inspecting both the current and previous generation, with the possibility of accessing individual solutions on demand. The interviews with the participants suggest that the interactive search was effective in exploring new areas of the search space, i.e. finding test inputs not usually conceived by humans. Their last work presents a new version of the iSBST tool with fault detection capabilities [15].

IV. Problem analysis

A. Limitations

As shown in Section III, different studies have assessed the effectiveness and acceptance of test generation tools in the past. In the experiments by Almasi et al., where both EvoSuite and Randoop were applied to find known faults in a real financial application [7], the tools struggled to find *hard* faults (i.e. those that require special test inputs) and *challenging* faults (i.e. those that additionally require satisfying complex conditions and constructing complex objects). As an example of a hard fault shown in [7], consider the faulty code excerpt in Listing 1, where the property *"inv_min"* (line 8) is an invalid key in the properties file. To detect this fault, the test case requires two specific strings (enumerated values "`POSITIVE`" and "`LOW`") to satisfy the two conditions that surround the fault (lines 3 and 5). In the aforementioned study, EvoSuite and Randoop were able to generate test cases to detect respectively 48% and 12% of these hard faults, and none of those judged as challenging. In contrast, it would be straightforward for any programmer to provide the strings to satisfy both conditions, thereby quickly reaching the faulty statement.

More importantly, most participants of the study were concerned about the readability of the automatically-generated tests, and even said that they preferred the test data and meaningful assertions used in the manually-written tests [7]. Interestingly, the participants said that, even though the generated assertions were not validating useful data, they could be easily transformed to become more appropriate. The findings in this study are in line with the conclusions drawn from the controlled experiment by Shamshiri el al. [11]. This study evidenced both that testers find automatically generated tests less readable because they do not resemble realistic scenarios

in general, and that they are more confident about their actions in the presence of manually-generated tests.

```
1   ...
2   List<Double> list = new ArrayList<Double>();
3   if(param3.equalsIgnoreCase(Enum1.POSITIVE.getValue())) {
4     for (int i=1; i<=param1*12; i++) {
5       if (param4.equalsIgnoreCase(Enum2.LOW.getValue())) {
6         //Faulty statement
7         list.add(i,Math.pow((1+Double.valueOf(
8                   PropertyReader.getProperty("inv_min"))),
9                   (Double.valueOf(1)/Double.valueOf(12)))-1);
10      }
11    }
12  }
13  ...
```

Listing 1. Example of a *hard* fault identified by Almasi et al. [7]

In summary, we have detected two major limitations in current search-based test generation tools that should be tackled:

- *Detection power*: Tools are not able to detect most of the hard-to-detect faults, i.e. those that require special test inputs or satisfying complex conditions.
- *Readability*: Software engineers cannot easily understand the purpose of the generated tests, e.g. how they map to the software functionalities.

B. Requirements

Given the limitations described in Section IV-A, we identify three requirements to be incorporated into the solution:

> ***Requirement 1:*** *Test generation is a white-box activity, which targets the internal structure of the system under test. White-box testing requires a broad knowledge of the source code of the classes under test.*

We consider a tester-centered approach, since only a tester possesses the necessary technical knowledge of the design and implementation of the program. In this sense, testers should be able to provide their skills and assess the generated solutions to reach meaningful test cases.

> ***Requirement 2:*** *Testers should be able to successfully incorporate their knowledge and preferences to the test generation process.*

Interactivity can allow testers to contribute to the automated generation of tests and to guide the evolutionary search towards better test suites. In addition, interactivity should allow adapting the process to the testing practices of the company and to the particular system design by means of a general-purpose test generation tool.

> ***Requirement 3:*** *Search-based test generation tools should be able to cope with two major limitations of current tools:* detection power *and* readability.

An evolutionary and interactive approach that offers diverse interaction opportunities can help deal with both detection power and readability. To improve the former limitation, the interaction mechanism should be oriented to meet a test criterion (e.g. some actions could help the search to reach hard-to-cover areas of code). Also, to improve readability, other interaction opportunities should be focused on assessing and increasing the acceptance of the generated tests (e.g. actions to enhance the readability could be related to the test sequence of method calls or the strings used in those invocations).

V. Putting the Human in-the-Loop for Test Generation

Interactive optimization has to be adapted for the test generation problem in order to face the aforementioned requirements. Therefore, we explore and determine the most suitable interactive options to improve detection power and readability. Based on the general components that guide the design of iSBSE proposals (see Section II-B) and relevant studies on interactive SBST, we propose specific options to introduce interactivity in search-based test generation. In particular, we focus on mutation testing, since it is a well-established testing technique that has been successfully applied to object-oriented programs. In addition, mutation testing has been used to drive search-based approaches [5]. Notice that most of these ideas are easily generalized to satisfy further test coverage criteria.

As aforementioned, SBST tools like EvoSuite can work with candidates at the level of test cases (only one coverage goal is considered at a time) or whole test suites (all coverage goals are optimized together). In what follows, we generally use the term "candidate" to indicate that an option is applicable to both approaches. This is possible because, if necessary, individual test cases could be selected from the whole test suites before an interaction takes place (e.g. showing those test cases that are closer to reach a particular mutant according to their individual fitness value).

a) Type of algorithm: Combining human-based evaluation and human-guided search seems to comply with the two purposes stated in the previous section. The former focuses on the subjective assessment of the tests, so it is more related to readability. The latter looks for benefiting from tester's coding abilities in order to cover hard faults. At this point, we should clarify that it is not expected that the tester will provide a complete and definitive solution, but to contribute in those aspects that remain challenging for the search algorithm.

b) Type of feedback: All user actions enumerated in Section II-B are applicable to the test generation process. For each one, Table I provides a list of specific options and whether they would improve detection power (D), readability (R) or both. Focusing on evaluation, the tester could assign a score to candidates with respect to detection capability or readability. Other options are rewarding/penalizing certain statements observed in the code, and specifying trade-offs if multiple objectives are pursued. Apart from evaluation, directly manipulating candidates would serve to order statements or edit parameters with the aim of providing values that could reveal a particular mutation. The comparison and selection of candidates can be used to discern between tests perceived as similar by the algorithm. All these actions do not only

TABLE I
TYPE OF FEEDBACK DURING INTERACTION FOR THE TEST GENERATION PROBLEM

ID	Types of feedback		D	R
1	Category	Candidate evaluation		
	Goals	- Give opinion on how close the candidate is to detect a non-detected mutant. - Rate the extent to which the candidate looks "human-written". - Reward/penalize certain sequences of method calls in new candidates in the next generations.		
	Options	1.1 *Assessment:* Assign a score for detection capability to candidates.	✓	
		1.2 *Assessment:* Assign a score for readability to candidates.		✓
		1.3 *Reward/penalization:* Indicate a sequence of lines to reward and/or penalize.	✓	✓
		1.4 *Weights:* Set trade-offs among testing criteria by re-weighting objectives.	✓	
2	Category	Candidate modification		
	Goals	- Incorporate technical knowledge to detect challenging faults. - Incorporate practical knowledge to generate meaningful candidates.		
	Options	2.1 *Modification:* Edit arguments of method calls.	✓	✓
		2.2 *Modification:* Add a complex object or data structure.	✓	
		2.3 *Modification:* Indicate methods that should be combined in the candidate.	✓	✓
		2.4 *Modification:* Include a complex assertion.	✓	
3	Category	Candidate selection		
	Goals	Choose among different candidates.		
	Options	3.1 *Comparison*: Choose preferred candidates among those that detect the same mutant.	✓	
		3.2 *Comparison*: Choose preferred candidates based on readability.		✓

TABLE II
INTERACTION SCHEDULING OPTIONS FOR THE TEST GENERATION PROBLEM

ID	Interaction scheduling		D	R
4	Category	Adjustment of interactive time and frequency of interaction		
	Goals	- Interact when the search has stalled after several generations trying to detect the fault. - Interact to give opinion about the readability of candidates.		
	Options	4.1 *Adaptive*: The algorithm decides to ask for feedback after n trials.	✓	
		4.2 *Adaptive*: Interaction starts once a minimum coverage has been reached.		✓
		4.3 *Adaptive*: The algorithm pauses after each mutation is detected.		✓
		4.4 *Fixed*: Interaction is scheduled at regular intervals.	✓	✓
		4.5 *On demand*: The user stops the search at his/her discretion.		✓
5	Category	Selection of solutions		
	Goals	- Show best candidates found so far. - Show several candidates that detect the same fault.		
	Options	5.1 *Best*: One or top n candidates according to their fitness.	✓	✓
		5.2 *All*: All candidates that detect the same target mutant.		✓
		5.3 *Specific*: Other criteria, e.g. number of detected mutants.	✓	✓
6	Category	Level of solution detail		
	Goal	Make the tester aware of the context required to inspect candidates appropriately.		
	Options	6.1 *Complete:* Show the whole candidate.		✓
		6.2 *Contextualized:* Test case together with the lines affected by the mutation.	✓	

allow the tester to help the algorithm cope with hard-to-detect mutants, but also adapt the code to his/her programming style.

c) Interaction schedule: If the tester has to intervene too frequently, this may cause fatigue. In the case of mutation testing, it would be unfeasible for the tester to review all possible mutants, as well as to discriminate equivalent ones, i.e. those mutants that do not change the functional behavior of the program. Table II summarizes the proposed methods for the adjustment of the interaction time and the frequency, the selection strategy and the level of detail. Firstly, the frequency can be either fixed (the system stops at regular intervals of time or iterations) or adaptive. For the latter option, the frequency might be determined by the algorithm based on some internal measures (e.g. generations without improvement) or depending on fault-detection milestones (e.g. every time a mutant is killed). Also, the start of the interactions could also be adjusted based on the coverage achieved by the search to avoid too early revisions. Another option is to let the tester make the decision according to the observed search progress. Secondly, an appropriate selection mechanism is required to show the tester a sample of representative tests. A common mechanism is to select the best individuals found so far [13]. However, we suggest that more specific criteria could be beneficial too, such as showing all the individuals that detect the same mutant, or those that detect a higher number of mutants besides the target mutant. Finally, apart from showing the candidates to review, they could also be accompanied with a particular mutation. The possibility to include such a *context* had not been considered in previous iSBSE studies [13], in which candidate solutions were either totally or partially visualized.

d) Information integration: Table III lists the available options concerning information integration. Notice that each type of feedback (see Table I) will influence on different steps of the algorithm. Subjective evaluation has to be reflected

TABLE III
INFORMATION INTEGRATION STRATEGIES FOR THE TEST GENERATION PROBLEM

ID	Information integration		D	R
7	Category	Information lifetime		
	Goals	- Decide for how long the tester's opinion influences the search. - Establish the possible scope of each type of feedback.		
	Options	7.1 *Mutation-based*: Detection and readability scores belong to a mutation-test case pair.	✓	✓
		7.2 *Mutation-based*: Code modifications are only valid for a mutation-test case pair.	✓	
		7.3 *Short-term*: Modified or preferred code lines are considered in other candidates.	✓	
		7.4 *Short-term*: Readability scores are kept during the search and reused in other candidates.		✓
		7.5 *Long-term*: Tester's preferences are saved for future executions.	✓	✓
8	Category	Information validity		
	Goal	- Determine whether the tester's decisions can be modified along the search.		
	Options	8.1 *Permanent*: The tester's feedback remains unaltered in the next iterations.	✓	✓
		8.2 *Flexible*: Scores and code modifications (parameters, asserts and method sequences) can be revisited.	✓	✓

somehow in the primary (fitness function) or secondary objectives, whereas modifications have a direct impact on the genotype of candidates. Sometimes, both actions have a similar lifetime, referred as *mutation-based*, since the feedback will be usually referred to a particular solution and a mutant. On other occasions, the opinions can be applicable to other candidates with a similar structure, thus propagating tester's judgment throughout the search in a *short-term* fashion. Furthermore, it would be interesting to derive rules capturing his/her programming preferences to be considered in future executions (*long-term*). Focusing on information validity, the simplest option consists in maintaining the tester's decisions along the whole search process. Implementing a mechanism tracking previous changes would be a more flexible alternative. Due to the particularities of mutation testing, decisions made for a pair solution-mutant might result in other mutants being killed or near to be detected. The algorithm is expected to use this information to plan which solutions will be shown next, so that the tester keeps focused on similar test cases.

VI. A PROOF OF CONCEPT

Section V presented a general model for interactive test generation, which has to be particularized depending on the goals pursued (detection, readability or both). To show how these concepts can be put into practice, we address a specific scenario focused on the subjective assessment of test suite readability. In this proof of concept, the tester will be asked to evaluate the readability of the most promising candidates (i.e. those with the highest coverage). The GA for test suite generation in EvoSuite is used as the base implementation, making some modifications and adding some parameters (see Section VI-A). Next, we describe a step-by-step execution of the interactive process. For the interested reader, a website[2] is publicly available, including additional technical details, screenshots and the generated solutions.

A. Including interactive options in EvoSuite

In this section, we describe the interaction options selected for each of the categories explained in Section V, as well as how they have been implemented in EvoSuite. As we are going to focus on readability assessment, human-based evaluation is the type of algorithm considered in this proof of concept. Next, we focus on the *type of feedback*, the *interaction schedule* and the *information integration*.

1) Type of feedback: We choose the subjective assessment of readability through scores as the type of feedback to be provided (option 1.2 in Table I). A new secondary objective is defined in EvoSuite so that candidates with high readability score according to the tester's perception are promoted without negatively impacting coverage. In our example, the rating scale is pre-established with a range 0-10. Readability scores will be then used to compare candidate test suites with the same fitness (aggregation of coverage criteria) in order to break ties during one of the following steps:

- **Sorting**: When a new generation has been produced, the population is sorted by the fitness value. The candidates at the top positions will have a higher probability to be selected for reproduction. In our example, the readability score serves to decide the relative ordering between candidates with equal fitness.
- **Replacement**: When offspring solutions are generated, their fitness values are compared with those of their parents. If the best descendant is less valuable than its best parent, offspring are discarded; otherwise, they are included in the next generation. In our example, a tie between the best child and its best parent can be solved by the tester giving an opinion about their readability.

We include a parameter (`When_to_revise`) in the GA implementation to set the step when an interaction will happen (sorting, replacement or both). Notice that, apart from candidate evaluation, the use of readability scores to break ties indirectly implies that option 3.2 in Table I is supported too.

2) Interaction schedule: Here we detail the decisions made regarding the adjustment of interaction time, the frequency of interaction, the selection of solutions and the level of solution detail (see Table II).

Firstly, we opt for an adaptive interaction process where the algorithm shows candidates after a minimum coverage has been reached (option 4.2 in Table II), a condition monitored during the evolution. More specifically, we de-

[2]https://www.uco.es/SEBASENet/CEC2021

fine a new parameter (`Revise_after_percentage_of_coverage`) to specify the percentage of all the goals that the best candidate found so far has to cover before enabling the new secondary objective. After the first interaction, the next ones are scheduled at a fixed frequency (option 4.4 in Table II). Similarly, a new parameter (`Revise_frequency`) is defined to configure the number of generations that have to elapse between interactions.

The selection of solutions depends on the step in which interaction happens. If the tester takes part in the sorting procedure, only a small subset of solutions should be selected to avoid information overload. Consequently, in our implementation, we limit the evaluation to the best candidates (option 5.1 in Table II), i.e. test suites tied with the highest fitness in the population. Recall that the candidates placed in the top positions will have more impact on the creation of the next generation; therefore, this decision helps to make the most of the information requested to the tester. As for interaction during replacement, the tester only has to revise the best child and the best parent —a kind of specific criterion (option 5.3 in Table II). Two parameters are included in the GA to add more control to the selection of candidates. On the one hand, we allow setting the maximum number of times the user is willing to interact during the search, either for sorting (`Max_times_sort`) or replacement (`Max_times_replace`). On the other hand, we allow limiting the maximum number of candidates from the population to review in one interaction (`Percentage_to_revise`). This parameter is only considered when the number of tied candidates surpasses this percentage. In that case, the selection is made at random.

Focusing on the level of solution detail, complete test suites are saved into files for tester's inspection (option 6.1 in Table II). However, it should be noted that the internal representation of the test suites in EvoSuite is different from their final appearance. To avoid large sequences of statements, a minimization process is carried out after the search to remove irrelevant statements. In this line, the selected candidates are minimized before being presented to the tester. Therefore, he/she will see the test suite as it would appear at the end of the search. If two test suites are equal after applying the minimization, they are grouped to avoid showing repeated information. Notice that minimization is a costly procedure, so we apply it once candidates have been selected instead of running it for all individuals with the best fitness.

3) Information integration: We adopt a short-term lifetime for readability scores in our implementation (option 7.4 in Table III). With this option, the provided scores will remain during the whole search and can be transferred to other test suites sharing the same minimized version. To support this, an archive of previously evaluated test suites —in its minimized form— is created. As for the information validity, a flexible mechanism (option 8.2 in Table III) is implemented to allow the tester to change previous assessments if desired. A new parameter (`Revisit_candidates`) is defined to specify whether already valued candidates will be shown in case they appear in the search again. When this parameter is enabled, the tester is informed about the previous evaluation and can change it. Otherwise, candidates are excluded from visualization and receive the value stored in the archive.

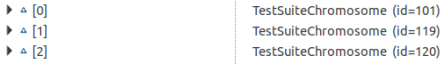
B. Step-by-step illustrative execution

We execute EvoSuite with the newly incorporated interactive features under the Eclipse IDE in order to generate a test suite for the class *ATM* (from the EvoSuite's tutorial) with a budget of 50 generations. We set the interaction-related properties as follows: `When_to_revise`: sorting; `Max_times_sort`: 3; `Percentage_to_revise`: 10%, i.e. 3 out of 30 candidates (the population size); `Revise_after_percentage_of_coverage`: 88%; `Revise_frequency`: 10 generations; and `Revisit_candidates`: false. The rest of EvoSuite parameters are left with their default values.

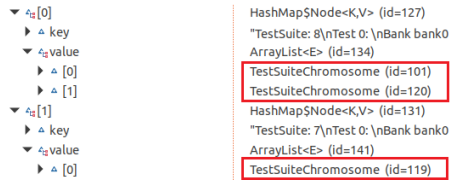
The following milestones can be remarked in the execution:

1. Start of the search until secondary objective activation: The search starts with the new secondary objective disabled. Hopefully, the fitness of the candidates –i.e. their coverage of goals– will be increased overall with each new generation. When the coverage of the best candidate is greater or equal to `Revise_after_percentage_of_coverage`, the secondary objective is enabled. In our example, this happens at the 15^{th} generation:

2. Selection of candidates: In our example, 26 out of 50 candidates present the same best fitness. However, according to `Percentage_to_revise`, only 3 of them can be selected for revision. These three candidates are then chosen at random:



3. Minimization: The selected candidates are minimized. In our example, two of the three candidates present exactly the same minimization. Therefore, they are grouped together and only two test suites are shown to the tester:



4. Interaction with the tester for readability assessment: At this point, EvoSuite is prepared for an interaction. The two minimized versions are saved to external files and the execution is paused to wait for the tester's feedback. In our example, the second minimized test suite receives a better score (8) than the first one (6).

```
IT IS TIME TO INTERACT! Go to the folder EvoSuite/CEC/InteractionSort-0.
After that, revise and provide a legibility value for each test suite.
```

```
Legibility for test suite 0:
6
Legibility for test suite 1:
8
```

5. Information integration and sorting: The readability scores are transferred to their respective candidate test suites, including those in the same minimization group. These candidates are re-ordered in the population going from highest to lowest readability score. In our example, the test suite with *id*=119 has the best score (*readabilityValue*=8), and, therefore, is placed at the first position:

6. Rest of the search: The secondary objective is disabled after that, and it will not be enabled again until `Revise_frequency` generations have passed. In our case, the following interactions are scheduled at iterations 25, 35 and so on. Whenever the search has not exhausted `Max_times_sort`, it will activate the secondary objective at those iterations, repeating steps 2-5 if it is necessary to break new ties. In our case, one additional interaction was requested before the end of the search. The test suite with the highest readability score is the one returned as the optimal solution with a final coverage of 95% of the goals.

VII. Conclusion

This paper explores how interactive optimization can be applied to take the most of both manual and search-based testing. Our proposal consists of letting testers have a direct influence on the test case design by providing new information to guide the search. As a result, readability becomes an explicit goal, while enhancing the capability of tools to detect challenging faults. The analysis of the different alternatives on how interactivity can be incorporated will serve to guide future developments. As a first approximation, we have implemented several interaction options in the well-known EvoSuite tool to support subjective readability assessment.

In the future, we plan to further analyze the viability of this proposal, as well as other implementations oriented to hard-fault detection, by means of empirical studies. We believe involving industry software testers could lead to revealing more faults while reducing the size of test suites. We hypothesize that this approach could also be applied at other testing levels, such as GUI or integration testing.

References

[1] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*, 3rd ed. Wiley Publishing, 2011.

[2] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, A. Bertolino, J. J. Li, and H. Zhu, "An orchestrated survey of methodologies for automated software test case generation," *J. Syst. Softw.*, vol. 86, no. 8, pp. 1978–2001, 2013.

[3] P. McMinn, "Search-based software test data generation: A survey: Research articles," *Softw. Test. Verif. Rel.*, vol. 14, no. 2, pp. 105–156, 2004.

[4] S. Wappler and J. Wegener, "Evolutionary unit testing of object-oriented software using a hybrid evolutionary algorithm," in *IEEE Int. Conf. on Evolutionary Computation*, 2006, pp. 851–858.

[5] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *IEEE Trans. Softw. Eng.*, vol. 38, no. 2, pp. 278–292, 2012.

[6] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg, "Does automated unit test generation really help software testers? A controlled empirical study," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 4, pp. 23:1–23:49, 2015.

[7] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds, "An industrial evaluation of unit test generation: Finding real faults in a financial application," in *Proc. 39th Int. Conf. on Software Engineering: Software Engineering in Practice Track*, 2017, pp. 263–272.

[8] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for Java," in *Companion Proc. 22nd ACM Conf. on Object-Oriented Programming Systems and Applications*, 2007, pp. 815–816.

[9] A. Arcuri, "An experience report on applying software testing academic results in industry: we need usable automated test generation," *Empir. Softw. Eng.*, vol. 23, no. 4, pp. 1959–1981, 2018.

[10] J. M. Rojas and G. Fraser, "Is search-based unit test generation research stuck in a local optimum?" in *IEEE/ACM 10th Int. Workshop on Search-Based Software Testing*, 2017, pp. 51–52.

[11] S. Shamshiri, J. M. Rojas, J. P. Galeotti, N. Walkinshaw, and G. Fraser, "How do automatically generated unit tests influence software maintenance?" in *IEEE 11th Int. Conf. on Software Testing, Verification and Validation*, 2018, pp. 250–261.

[12] M. B. Cohen, "The maturation of search-based software testing: Successes and challenges," in *12th Int. Workshop on Search-Based Software Testing*, 2019, pp. 13–14.

[13] A. Ramírez, J. R. Romero, and C. L. Simons, "A systematic review of interaction in search-based software engineering," *IEEE Trans. Softw. Eng.*, vol. 45, no. 8, pp. 760–781, 2019.

[14] B. Marculescu, R. Feldt, R. Torkar, and S. Poulding, "An initial industrial evaluation of interactive search-based testing for embedded software," *Appl. Soft Comput.*, vol. 29, pp. 26–39, 2015.

[15] B. Marculescu, R. Feldt, R. Torkar, and S. M. Poulding, "Transferring interactive search-based software testing to industry," *J. Syst. Softw.*, vol. 142, pp. 156–170, 2018.

[16] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege, "A systematic review of the application and empirical investigation of search-based test case generation," *IEEE Trans. Softw. Eng.*, vol. 36, no. 6, pp. 742–762, 2010.

[17] G. Gay, "The fitness function for the job: Search-based generation of test suites that detect real faults," in *IEEE Int. Conf. on Software Testing, Verification and Validation*, 2017, pp. 345–355.

[18] D. Meignan, S. Knust, J.-M. Frayret, G. Pesant, and N. Gaud, "A review and taxonomy of interactive optimization methods in operations research," *ACM Trans Interact Intell Syst.*, vol. 5, no. 3, 2015.

[19] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Comput. Surv.*, vol. 45, no. 1, 2012.

[20] E. Enoiu, D. Sundmark, A. Čaušević, and P. Pettersson, "A comparative study of manual and automated testing for industrial control software," in *IEEE Int. Conf. on Software Testing, Verification and Validation*, 2017, pp. 412–417.

[21] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges," in *30th IEEE/ACM Int. Conf. on Automated Software Engineering*, 2015, pp. 201–211.

[22] G. Gay, "One-size-fits-none? Improving test generation using context-optimized fitness functions," in *IEEE/ACM 12th Int. Workshop on Search-Based Software Testing*, 2019, pp. 3–4.

[23] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer, "Modeling readability to improve unit tests," in *10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 107–118.

[24] E. Daka, J. Campos, J. Dorn, G. Fraser, and W. Weimer, "Generating readable unit tests for Guava," in *7th Int. Symposium on Search-Based Software Engineering*, 2015, pp. 235–241.

[25] B. Marculescu, R. Feldt, and R. Torkar, "A concept for an interactive search-based software testing system," in *4th Int. Symposium on Search Based Software Engineering*, 2012, pp. 273–278.