TESRAC: A Framework for Test Suite Reduction Assessment at Scale

João Becho
LASIGE & FCUL
University of Lisbon
fc42103@alunos.fc.ul.pt

Frederico Cerveira
University of Coimbra, CISUC,
Department of Informatics Engineering
fmduarte@dei.uc.pt

João Leitão
NOVA LINCS & FCT
NOVA University of Lisbon
jc.leitao@fct.unl.pt

Rui André Oliveira†
NOVA LINCS & FCT
NOVA University of Lisbon
rac.oliveira@fct.unl.pt

2022 IEEE Conference on Software Testing, Verification and Validation (ICST) | 978-1-6654-6679-0/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICST53961.2022.00028

***Abstract*—Regression testing is an important task in any large software project, however as codebase increases, test suites grow and become composed of highly redundant test cases, thus greatly increasing the time required for testing. To solve this problem various test suite reduction tools have been proposed, however their absolute and relative performance are unclear to their prospective users, since there is a lack of a standardized evaluation or approach for choosing the best reduction tool. This work proposes TESRAC, a framework for assessing and comparing test suite reduction tools, which allows users to evaluate and rank a customizable set of tools in terms of reduction performance according to criteria (coverage, dimension, and execution time), and which can be configured to prioritize specific criteria. We used TESRAC to assess and compare three test suite reduction tools and one test suite prioritization tool that has been adapted to perform test suite reduction, across eleven projects of various dimensions and characteristics. Results show that a test suite prioritization tool can be adapted to perform a adequate test suite reduction, and a subset of tools outperforms the remaining tools for the majority of the projects. However, the project and test suite being reduced can have a strong impact on a tool's performance.**

***Index Terms*—software testing, test suite reduction, test suite minimization, test case prioritization, evaluation**

I. Introduction

Software testing is the de facto approach for uncovering software faults, also known as software bugs, and is nowadays an essential step during software development. A software test consists of two basic elements: an input that executes in the system under test (SUT) and a definition of the expected outcome [1]. Test suites are a set of predefined test cases (i.e. software tests), which are executed sequentially in order to uncover faults in the SUT. Software testing is a common practice in industry, with many companies nowadays integrating testing as part of the development and evolution cycles, for instance by resorting to continuous integration [2].

While companies have to manage a plethora of software projects, including thousands of commits made to the source code on a daily basis, at the same time they need to maintain an appropriate software testing infrastructure. One task of such an infrastructure is to manage all the test suites that are executed before accepting any changes to the software or before publishing new releases. Companies usually perform these tests to check if changes in existing modules behave as expected and that the unchanged modules did not experience an impact in functionality (i.e. creating a "regression") [3]. This particular process is referred to as regression testing.

A common challenge in this context is ensuring that test suites remain up-to-date (i.e., new test cases are added and outdated test cases are removed) and that these can be efficiently executed, while ensuring the desired code coverage independently of frequent changes to the software. As projects grow in terms of lines of code, the number of test cases inevitably increases, resulting in some lines of code (i.e., execution paths) being covered by multiple test cases. This redundancy creates inadvertently large test suites that are difficult to maintain and whose execution might become too costly – both in terms of time and monetary cost – hindering the evolution of software. It has been previously reported that testing software containing approximately $20,000$ lines of code requires seven weeks for the entire test suite to run [4], and also that, in large and complex software infrastructures, removing one redundant test case may save thousands of dollars in testing resources [3].

In recent years two key techniques have been associated with regression testing: i) test suite reduction (TSR) and ii) test case prioritization (TCP) [5]. The goal of test suite reduction (also known as minimization) is to find a reduced test suite by eliminating redundant test cases according to a number of criteria [6]; test case prioritization aims to define the ordering of test cases based on a particular goal, such as maximizing code coverage or to improve fault detection effectiveness [6]. Although these two techniques have fundamentally different goals, a TCP tool can rank test cases and a reduced test suite can be defined by discarding the less relevant test cases. Software testers have at their disposal a large number of tools that have been proposed in recent years to lower the burden of maintaining test suites small and efficient.

Despite the cost benefits of TSR, software testers who work in safety-critical systems are often reluctant to omit test cases due to concerns that the fault detection capabilities of the test suite might be affected [3]. This is understandable and, partially, explainable due to the lack of tools that allow to effectively and systematically assess and compare existing TSR tools. This leads testers to be required to download each TSR tool, install and configure its dependencies, and execute

†: Part of this work was conducted while Rui André Oliveira was affiliated with the LASIGE laboratory and the University of Lisbon.

978-1-6654-6679-0/22/$31.00 ©2022 IEEE
DOI 10.1109/ICST53961.2022.00028

Authorized licensed use limited to: b-on: UNIVERSIDADE NOVA DE LISBOA. Downloaded on February 05,2023 at 14:32:34 UTC from IEEE Xplore. Restrictions apply.

each tool for each test suite for every project in which they are involved. Even after going through these steps, testers will still have to select the most appropriate tool for each project considering a plethora of criteria: the size of the reduced test suite, the code coverage, and the time taken to execute the test suite, which can be cumbersome and error-prone.

Existing studies surveying TSR tools focus on comparing the technical aspects of the employed algorithms [5], [6], disregarding an empirical evaluation of their effective performance in real-world projects. To the best of our knowledge, there is not a well-defined approach for evaluating the effectiveness of TSR tools. A framework for assessing and comparing TSR tools should: i) enable software testers or practitioners to painlessly select the best TSR tool for a particular project; ii) enable researchers in the field of TSR to understand how their tool competes with the state of the art and promote an healthy competition; and iii) enable practitioners to effectively derive efficient and adequate test suites for their projects. We note that the existence of such framework can significantly contribute to promote an healthy competition between existing and upcoming TSR techniques/tools and advance the current state of the art.

In this paper we present TEst Suite Reduction Assessment at sCale (TESRAC), a framework for assessing and comparing the effectiveness of TSR tools. TESRAC applies a novel approach that collects different characteristics related with the effectiveness and performance of the tools on test suite reduction. These characteristics include *Dimension*, *Time Taken to Execute* and *Coverage*, which are used to compute a *Reduction Score* using a multi-criteria decision making technique. Our current TESRAC implementation supports assessing the TSR efficiency of three Java tools: EvoSuite [7], Testler [8], and Randoop [9]. We also included support for Kanonizo [10], a known TCP tool, where our framework adds a layer to discard the test cases considered less relevant. In this way we treat Kanonizo as a TSR tool, and see how it compares to the native TSR tools. TESRAC internal architecture is modular and extensible, supporting the integration of new TSR and TCP tools. TESRAC is available as an open-source tool [1], enabling third-party TSR developers to integrate their solutions, while also enabling third-party practitioners to extend its support to other testing scenarios. We used TESRAC to test 11 different projects featuring test suites with distinct levels of complexity, ranging from a dozen to hundreds of test classes, as to show its usefulness.

The main contributions of this work include:

• We present TESRAC, the first framework for assessing and comparing the effectiveness of TSR tools. TESRAC gathers multiple criteria, namely software quality metrics (e.g., coverage metrics, mutation score), performance of the execution process, and dimension of the reduced test suites, combining them using a multi-criteria decision making approach. Software testers can adjust and fine-tune TESRAC, to balance the importance of each criterion according to different requirements before computing of the final reduction Score.

• We have implemented TESRAC to evaluate Test Suite Reduction and Test Suite Prioritization tools for Java. Our implementation follows a modular architecture, enabling it to be extended to include new reduction or prioritization tools and new software quality metrics as sub-criteria.

• We conducted and present the results for an experimental evaluation where TESRAC is used for assessing and comparing three test suite reduction tools: Testler, EvoSuite and Randoop, and one test case prioritization tool: Kanonizo, using 11 (open source) projects. Our results show that TESRAC can cutoff the less relevant test cases, thus enabling any prioritization tool to be transformed into a reduction tool.

This paper is organized as follows. Section II introduces related work. Section III presents an overview of TESRAC. Section IV details the experimental setup used in our evaluation. Section V reports our experimental results. Section VI discusses the key findings and limitations of our work. Finally, Section VII concludes this paper and discusses future work.

II. Related Work

There is a large selection of TSR based studies and tools that were surveyed in previous studies [5], [6]. These tools are available as commercial or free and open source licenses [6] and are presented on the next paragraphs.

ATAC [11] is a tool that analyses the coverage achieved by tests on C and C++ programs. This is achieved essentially through three steps: i) the instrumentation of the source-code to be tested, which is achieved via the ATAC compiler, to gather data regarding the program execution; ii) the execution of the tests, which effectively allow to gather relevant information; and iii) the coverage analysis, which gathers information about the tests, such as line and branch coverage, that will be used as input to the tester. One of the features highlighted by the authors is the reduction of regression tests set size by eliminating redundant tests.

RUTE-J [12] is an effort to solve some of the most common problems in randomized unit testing, such as the correct definition of arguments, whether they are scalar or complex, and specifying correct behavior of tests. To use RUTE-J, the user must write a Test Fragment Collection, which is essentially a Java class that extends a RUTE-J class. Once the Test Fragment Collection set up, RUTE-J UnitDriver is executed, which initializes a graphical user interface, where the user can generate test cases for the executed test fragment collection. When RUTE-J finds a failing test suite it halts the generation, giving the user the opportunity to minimize the failed test suite, discarding the unnecessary statements to achieve the fail state.

Randoop [9] is a tool that generates JUnit test cases while reducing failing test cases. To generate test cases, Randoop employs a technique that gathers feedback from the execution of tests as they are generated [9], so that it can generate better and more reliable tests. Randoop receives as input a JUnit test suite and tries to reduce every test that fails by analyzing each

[1]Source code available at https://github.com/racoq/TESRAC

Authorized licensed use limited to: b-on: UNIVERSIDADE NOVA DE LISBOA. Downloaded on February 05,2023 at 14:32:34 UTC from IEEE Xplore. Restrictions apply.

statement and provide a simplification. The simplifications made to a statement can be achieved by applying the following operations: i) if a statement is represented by null, it is removed; ii) replacing the right hand side expression with 0, false or null, according to the left hand side expression; iii) replacing the right hand side expression with a calculated value obtained from a passing assertion; and iv) removing the left hand assignment of a statement.

Open-SourceRed [13] is a framework that simultaneously implements TSR and TCP algorithms. It consists of two tools: Proteja and Modificare. Proteja is written in Java and collects coverage information by running the JUnit test suites associated with some software, outputting coverage reports. These are used as input to Modificare for running the reduction and prioritization algorithms. Modificare is written in R and reads the reports generated by Proteja to apply one of the six implemented TSR and TCP algorithms: random, adaptive random, greedy, hill climbing, simulated annealing, and genetic.

EvoSuite was developed by Fraser and Arcuri [7]. It implements search-based and mutation techniques to generate and reduce test suites as small as possible ensuring high coverage according to some criterion. While generating the tests, EvoSuite uses a search-based approach, considering a population of candidate solutions and generating new solutions by reproduction of the best individuals, according to a fitness function. In the case of EvoSuite, the candidate solutions are test suites, each of them with a set of test cases. The reproduction of two candidates is done by exchanging some of their test cases between each other. In addition, mutation in the candidates is also employed, by adding, removing, or changing individual statements or parameters in some test cases.

Testler [8] employs a fine-grained minimization technique to achieve Test Suite Reduction. It analyzes the behavior of the test cases at the statement level to infer a model that represents the relationship between the test statements and test states, which is the information about a test at the time of each statement executed, such as the defined variables, their values and the production method calls. Based on this model, Testler can detect any fine-grained redundancy between test cases in the same test suite, and reorganizes the suites so that it removes these redundancies.

JTOP [14] is a tool developed as an Eclipse plugin that helps the management of JUnit test suites by statically analyzing the software under tests and conducting test reduction, prioritization, and selection.

TOBIAS [15] is a web application that applies combinatorial test suite generation, to an input file from the user, written in a specification language to be interpreted by TOBIAS. This file describes the methods the test suite should test, and the inputs it should use to do it. TOBIAS generates these test suites in a combinatorial way, leading to really large test suites. To address this issue it allows to connect the generator of the test suite to "selectors" that choose a subset of the test suite based on various techniques or criterias. The subset of test suites are selected based on a stochastic approach.

TEMSA [16] is a web-based application where the user provides a XML file as input,that specifies the features and test cases that should be used to test a product. It considers cost measures, such as overall execution time, and effectiveness measures, such as fault detection capability or test minimization percentage, that can also be set by the user in the website. Based on this input, it then generates a set of XML files representing the minimized test suite.

Regarding Test Case Prioritization, excluding OpenSourceRed, there is one open source implementation worth mentioning. Kanonizo [10] re-orders test cases such that the ones likely to detect faults are executed first. It implements several algorithms materializing several methodologies namely: random search and genetic algorithm (search-based), random, greedy, additional greedy, and schwa, which operate at the test case level; marijan, huang, cho, and elbaum, which are history-based, meaning that they resort to an history file with information related to previous test executions.

The majority of existing tools and ongoing research in the context of test suit reduction are focused on Randomized Unit Testing (RUT) [6]. RUT reduction tools use approaches that generate distinct test inputs as to help exposing internal errors. The remaining classes of TSR are not considered for our study. From the RUT tools presented above Randoop, EvoSuite, Testler, and Kanonizo were integrated in our tool implementation and considered for empirical evaluation. We opted to select all tools that reduced JUnit-based test suites, which is the most widely used testing environment for Java-based applications. This requirement excluded most other existing tools. We have not integrated JTOP since it was developed as an Eclipse plugin and only provided in a binary form. We tried to find the source code but it was unavailable. We opted to include support to Kanonizo, where we manually perform the test suite reduction discarding the less relevant test cases.

III. The TESRAC Framework

This section presents TESRAC's internal architecture, which is composed of three main components (illustrated on the left side of Figure 1). The first component is named *Input Selection* and is used to select the project and the test suites for the execution on TESRAC. The second component is named *Reduction and Execution* and is responsible for reducing the test suites using the TSR and TCP tools, executing the test suites using a code coverage and reporting tool, and performing mutation testing over the projects and test suites. The third component is named *Processing and Output* and it gathers relevant metrics from the previous component, which are used to characterize the reduction and the mutation testing and are then fed into a Multi-Criteria Decision Making (MCDM) technique to produce the final score. In this section, we detail the internal operation of each of these components (illustrated on the right side of Figure 1).

A. Input Selection Component

When the framework starts, it expects the user-specified project as its input. Inside the *Input Selection* component, two

Authorized licensed use limited to: b-on: UNIVERSIDADE NOVA DE LISBOA. Downloaded on February 05,2023 at 14:32:34 UTC from IEEE Xplore. Restrictions apply.

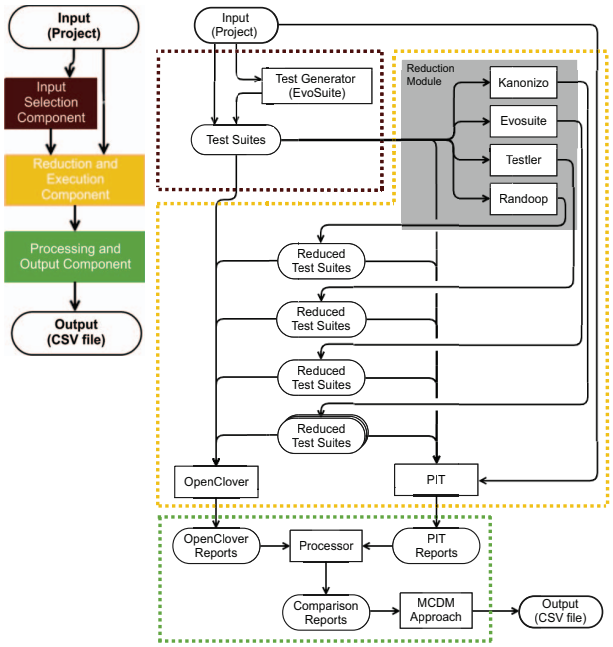


Fig. 1: An overview of the TESRAC Framework.

possible workflows are available: (i) retrieve the test suites included by the project, or $ii)$ generate new test suites. The latter workflow can be useful when the project lacks a test suite or when the user wants to generate new test suites. In this workflow, TESRAC uses the **Test Generator** module that integrates with the popular test suite generation tool EvoSuite. In either case, the execution flow continues to the *Reduction and Execution* component.

B. Reduction and Execution Component

Inside the *Reduction and Execution* component lays the key modules that support the operation of TESRAC. After the test suite is collected in the previous component, TESRAC reduces it in the **Reduction Module**. This module includes support for a range of pre-integrated tools, namely: EvoSuite, Testler, Randoop, and Kanonizo. Since TESRAC adopts a flexible architecture, it is possible for anyone to extend it to support other TSR and TCP tools by writing simple adapters to interact with external tools.

While some of the mentioned tools have a clear application to TSR (such as Testler or Randoop), EvoSuite and Kanonizo had to be further modified and integrated as to be able to take advantage of their reduction capabilities. Since EvoSuite's primary goal is to generate test suites using genetic algorithms to create different test case combinations, and test suite reduction is only performed at a later stage, a wrapper class was written to directly take advantage of EvoSuite's reduction features and to integrate them into TESRAC. As a limitation, EvoSuite can only reduce self-generated test suites and cannot be applied to existing external test suites. Kanonizo is a TCP tool that was integrated into TESRAC by repurposing its prioritization mechanism into a mechanism capable of performing TSR. To do so, TESRAC enables a threshold (or cutoff value) to be set (in a configuration file) that will be used to define the percentage of less relevant test cases that should be removed from the prioritized list of test cases. TESRAC supports the usage of various of Kanonizo's prioritization algorithms, including *random*: the test suite is randomly ordered; *random search*: determines the important test cases using a fitness function; *greedy*: prioritizes test cases that cover the maximum number of lines; and *additional greedy*: similar to greedy, but excludes already covered lines. Furthermore, TESRAC supports simultaneously creating multiple reduced test suites for different cutoff values.

After the reduction task is performed, the full set of test suites (i.e., the test suites before and after the reduction) are supplied as input to a code coverage analysis tool. TESRAC has built-in support for OpenClover [17], a mature project that supports more than 20 built-in code metrics, which is used to process and gather code coverage metrics from the full set of test suites.

In the end, mutation testing is performed over the projects to run the full set of test suites to assess the mutation coverage, which measures the fault-detection capabilities of each test suite. PIT Testing [18] was chosen as the mutation testing tool to be integrated in TESRAC, since it is a popular and proven solution for Java. We note again that since the architecture of TESRAC is modular, both OpenClover and PIT can be easily replaced by similar tools.

C. Processing and Output Component

Using the information obtained from OpenClover and PIT, this component calls the **Processor** module, which generates a comparison report for each project and tool combination. Each of these reports (outputted as a CSV file) states, for every source class, test class and reduced test class, all of the code coverage metrics and mutation scores.

These reports are then fed to the **Multi-criteria Decision Making** module, which applies an approach inspired by the Analytic Hierarchy Process (AHP) technique [19]. This approach allows to weight multiple criteria for comparison and to produce a score that can assist in the decision making process of selecting a particular subject (e.g., a TSR/TCP tool).

To apply our approach a quality model was defined, identifying the main criteria and sub-criteria used for the computing the final score [19], which is depicted in Figure 2.

The MCDM quality model considers three criteria: Dimension, Time, and Coverage. In the context of the **Dimension** criterion, we consider two sub-criteria: file size of each test suite and the number of test cases. For the **Time** criterion, the only sub-criterion considered is the execution time for each test suite. For the **Coverage** criterion, the sub-criteria are branch coverage, total coverage and mutation coverage. The total coverage is calculated, based on the branches that are evaluated to true or false (at least once), and based on the statements covered and methods entered, as defined in

Authorized licensed use limited to: b-on: UNIVERSIDADE NOVA DE LISBOA. Downloaded on February 05,2023 at 14:32:34 UTC from IEEE Xplore. Restrictions apply.

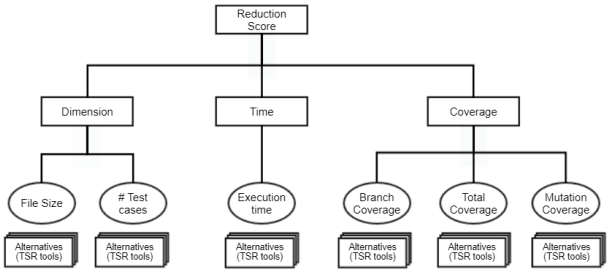


Fig. 2: MCDM Quality model employed in TESRAC.

OpenClover documentation [20]. Despite our choice of sub-criteria, which aimed to use metrics capable of representing the success of TSR (such as branch and total coverage [5], [6], both supported by OpenClover [17]), the approach is flexible and can be tailored to include other criteria and sub-criteria.

When the execution of TESRAC reaches the MCDM module, it loads the criteria from its configuration file and computes the priority for each sub-criterion as defined in AHP, according to the importance values attributed to each sub-criterion. These importance values should be defined by the user in the configuration file and, intuitively, should answer the following question: "How much more important is criterion X in relation to criterion Y?", according to the fundamental scale of importance, proposed by Saaty [21].

Normally the next step would be to compute the pairwise comparison matrix between the alternatives (the reduction tools) and their priorities, which would then be used to rank the tools. However, we decided to diverge from the AHP in this step, as we wanted the scores of each tool to be independent from one another. Instead, we use the sub-criteria priorities and the values measured for the sub-criteria for each tool to compute their respective scores. For each criterion, the values are a combination of the measurement of the related metric for the original test suites and the reduced one, in such a way that, for:

- Cost sub-criteria (such as file size, number of test cases and test suites execution time): we take $v_{original}$ and $v_{reduced}$ and calculate $(1 - \frac{v_{reduced}}{v_{original}}) * 100$. In the case that $v_{original}$ is 0, we define the whole value to be 0.
- Benefit sub-criteria (such as branch coverage, total coverage and mutation coverage): we take $v_{original}$ and $v_{reduced}$ and calculate $(\frac{v_{reduced}}{v_{original}}) * 100$. In the case that $v_{original}$ is 0, we define the whole value to be 0.

Once all these values are collected, we normalize inside of each (Project, Sub-criterion) pair, so that we can relate each value obtained for the different tools, for the same project and sub-criterion. We normalize using the formula $(\frac{v_i}{v_{max}})$, where v_i is the value obtained for a given tool, in the (Project, Sub-criterion) pair, and where v_{max} is the maximum value from the different values measured for this project in this sub-criterion by every tool.

The final step in getting the reduction score for a tool in a given project is performing a weighted sum between the sub-criterion's priorities (computed previously in the MCDM Approach module) and the normalized values measured for each of the sub-criterion in the given project. This ensures that the score for a given tool will always be a value between 0 and 1, with values closer to 1 being better. As such, we can rank the tools by sorting their scores in a descending order.

The output of the MCDM module (produced as a CSV file) provides the reduction scores that each tool achieved in each of the tested projects, as well as the total score computed for that tool. TESRAC creates $n + m \times k$ test suites for each of the original test suites of a particular project where n is the number of integrated tools (except for Kanonizo), m is the number of Kanonizo algorithms used and k is the number of cut off values defined.

Finally, we note that the user can configure TESRAC to execute in two modes: *full mode*, in which the flow of execution goes through all of the described components; and *analyze mode*, in which only the Processing and Output Component is executed. This mode is useful when there are already comparative reports generated by previous TESRAC executions and the user wants to evaluate the tools according to other criteria or by modifying the importance associated with sub-criteria, allowing for an easy re-evaluation.

IV. Experimental Setup

TESRAC was used to conduct an extensive experimental work where 11 different open source Java projects were compared. In this section we detail our testing environment, the test subjects, and the experiments.

A. Testing environment and used software

The experiment was executed across two different configurations of machines: a laptop with an Intel Core i7 7700HQ processor and 16Gb RAM executing Windows 10; and a cluster of machines, all equipped with Intel Xeon E5-2609 v4 processors and 32Gib RAM each executing Debian 10.

TESRAC was implemented using Eclipse IDE v4.8.0, using Java version 1.8.0_231 and integrates with the code coverage analysis tool OpenClover (v4.3.1) and the mutation testing tool PIT (v1.5.2). Our current implementation of TESRAC is fully integrated with EvoSuite (v1.0.7), Testler (v0.0.1), Randoop (v4.1.2), and Kanonizo (v0.0.1). In the case of EvoSuite, only the reduction process after the generation of the test suites is evaluated. It is also worth noting that, although Kanonizo is a test case prioritization tool and does not perform test suite reduction, TESRAC can perform cutoffs to the less relevant test cases, thus converting a TCP tool into a TSR tool.

B. Test subjects

In our experiments, we consider 11 different open-source Java projects. As the tools that TESRAC integrates reduce JUnit tests, the chosen projects feature JUnit test suites and use the Maven build system. We have selected projects that are representative of different domains, and also exhibit different characteristics, such as the size of the codebase and test suite. Table I reports the number of classes (between 10 and 637)

Authorized licensed use limited to: b-on: UNIVERSIDADE NOVA DE LISBOA. Downloaded on February 05,2023 at 14:32:34 UTC from IEEE Xplore. Restrictions apply.

and the number of test classes (between 2 and 350) for the projects that we have used in our evaluation.

TABLE I: Java open-source projects employed in our experimental work.

Project	Version	Dimension			# Dependents
		LOC	# Classes	# Test Classes	
commons-lang	3.7	78 000	151	172	121 866
commons-email	1.5	6 182	23	26	14 219
pmd-core	6.20	48 221	350	102	160
tudu-lists	3.0	4 497	49	20	0
lambdaj	2.4.1	8 325	95	61	3 813
jfreechart	1.6.0	221 637	637	350	6 213
java-library	2.1.0	36 298	437	157	9
crunch-core	0.15.0	37 954	297	185	6
crunch-kafka	0.15.0	2 449	14	16	6
tika-xmp	1.18	1 644	10	2	97
xml-sec	2.0.8	70 530	392	198	1 745

C. Experimental Methodology

The experiments reported here can be separated in two parts: i) generation, reduction, and analysis of test suites, and ii) computing of reduction scores.

For each of the tested projects, we have conducted two types of experiments: one using already provided test suites in each project - defined as **Normal runs** - and another using test suites generated by EvoSuite for each project - defined as **EvoSuite runs**.

Kanonizo was configured to use four prioritization algorithms (whose descriptions were previously presented): *Random*, *Random Search*, *Greedy*, and *Additional Greedy*. In addition to the algorithms, five cut-off values were used to determine the percentage of less relevant test cases to remove: 10%, 15%, 20%, 25% and 30%. This means that Kanonizo will act as 20 different tools, with each variation of the experiment being labeled as *(algorithm, cutOffValue)*. Each variation of Kanonizo that executed the Random algorithm was repeated 30 times to ensure the reliability of results. We note that other configurations produce fully deterministic results and hence were executed a single time.

Generation, reduction, and analysis of test suites: The experiment begins with the reduction (and generation, for the EvoSuite runs) of the test suites for each of the projects. Each of the original and reduced test suites was then used by OpenClover and PIT to retrieve a set of metrics - File size, number of test cases, test suites' execution time, total coverage, branch coverage, and mutation coverage. These represent the pre-defined metrics that will be used in the evaluation.

Calculating reduction scores: The values of the metrics retrieved from the reduced suites are combined with the values from the original suites, which are then used to compute a tool's reduction score for each project, using the MCDM approach and according to the configured importance of each metric. This allowed for different configurations of the metrics (sub-criteria of the MCDM), allowing for the evaluation of the integrated tools in different scenarios.

For each run (Normal and EvoSuite), the tools were evaluated in six different scenarios: one real-world scenario that tried to replicate a real-world need for managing and reducing a test suite, and five scenarios where we focus on one of the performance metrics.

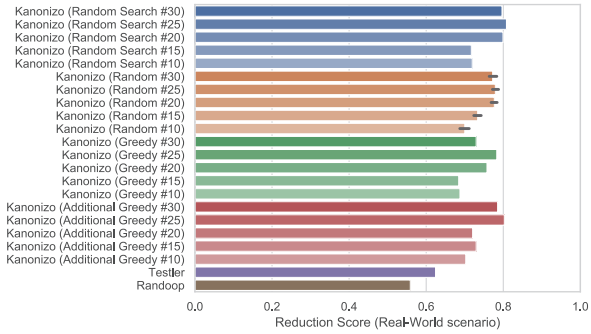
V. Evaluation

The experimental evaluation aimed at answering three research questions related to the viability and performance of the evaluated reduction and prioritization tools and the effect that the project being reduced can have on the performance. Thus this section is organized into three subsections that answer each one of these questions in turn.

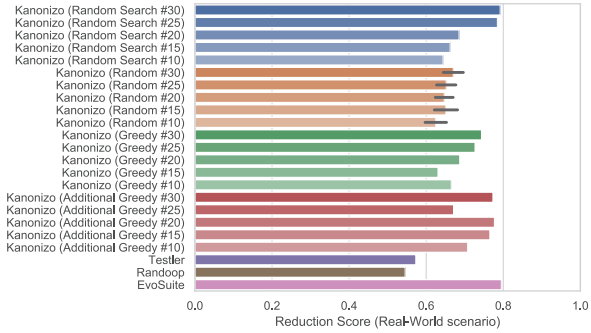
A. RQ1: Considering the pre-defined metrics, what is, in relative terms, the best TSR?

Figures 3a and 3b present the (final) reduction score obtained by every tool and all evaluated combinations of algorithms and cutoffs in a real-world scenario (where the scores of MCDM are distributed in a balanced manner but with a slight preference for higher coverage and lower execution time).

The top figure refers to experiments in the normal test scenario while the second refers to experiments in the EvoSuite scenario. Due to the non-deterministic nature of Kanonizo's Random algorithm, 30 independent executions were conducted, with error bars being presented in the figures reporting the standard deviation.



(a) Normal Run.



(b) EvoSuite Run.

Fig. 3: Reduction scores per tool on the real world scenario.

Considering the results in the normal test scenario, the best performing tools were Kanonizo with Random Search and a

Authorized licensed use limited to: b-on: UNIVERSIDADE NOVA DE LISBOA. Downloaded on February 05,2023 at 14:32:34 UTC from IEEE Xplore. Restrictions apply.

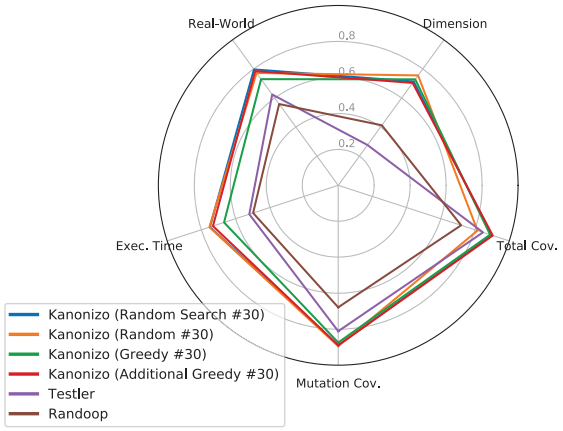
25% cutoff, with a reduction score of 0.808, Kanonizo with Additional Greedy and a 25% cutoff, with a score of 0.802 and Kanonizo with Greedy and a 25% cutoff, with a score of 0.783. Interestingly the ideal cutoff appears to be 25% in every algorithm, because results drops slightly when considering cutoff values above of below this. This suggests that after the 25% cutoff mark, additional test cases have little benefit to the coverage, only resulting in increasing the test suite size and execution time.

In the EvoSuite runs, EvoSuite was the best alternative, with a score of 0.794, but followed closely by Kanonizo with Random Search configured with cutoff of 30% and 25% with a results of 0.791 and 0.784 respectively. In these runs, the ideal cutoff for most Kanonizo algorithms appears to have moved to 30%, which suggests that the test suite being reduced will govern the ideal cutoff value. Interestingly, the Additional Greedy algorithm saw best performance at a 20% cutoff and worst performance at 25% cutoff. Randoop and Testler remained the two least effective tools.

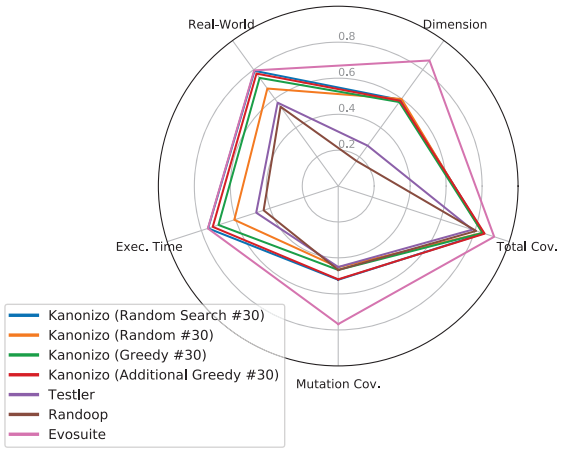
In both cases, the Random algorithm was the worst performer from all of Kanonizo's algorithms and further suffers from high deviation, given its non-deterministic nature.

Different configurations other than the real world scenario were considered. These purposefully focused on a specific criterion from the set of test suite dimension, coverage and execution time, by increasing the MCDM importance values for that criterion to a higher value and reducing the remaining scores to a lower value. The precise weights assigned to each scenario are shown in Table II. The idea behind having multiple scenarios is to encompass various choices that can be done by users of the framework and to represent specific applications (e.g., software with strong dependability requirements may prefer to prioritize mutation coverage to maximize software bug detection). Figures 4a and 4b present, for the Normal and EvoSuite test scenarios respectively, a spider chart where each axis refers to a specific scenario (real-world, mutation coverage focused, file size focused, total coverage focused, and execution time focused) and the reduction scores obtained by each tool for that specific scenario are plotted. The same cutoff value of 30% is plotted for all algorithms of Kanonizo and the line depicting the Random algorithm represents the average of all 30 runs.

These charts enable an easy verification of how the various tools perform when the focus of the test scenario changes. With regards to the Normal runs, Randoop showed the worst performance in all the scenarios except the dimension focused scenario, while Testler was the second worst performing tool. Kanonizo and its various algorithms showed the best performance independently of the scenario being considered. Kanonizo using the Random algorithm performed poorly in the scenario that focused on total coverage, even having a lower score than Testler, however it showed similar performance to the remaining alternatives when focusing on mutation coverage. The Greedy algorithm showed slightly worse performance on the scenario that focused on execution time of the test suite and in the real-world scenario.



(a) Normal runs.



(b) EvoSuite runs.

Fig. 4: Spider chart with reduction scores for various scenarios.

On the EvoSuite runs, EvoSuite appears as the best alternative in every scenario with good results in the scenarios that focus on mutation coverage and dimension. This means that EvoSuite is capable of reducing test suites to an high degree (i.e., creating reduced test suites with a low size) while maintaining very good mutation coverage performance. Randoop and Testler remain the worst performing tools, while the various Kanonizo configurations had a performance that is acceptable across all scenarios, but with results below that of EvoSuite in scenarios focused on total coverage, mutation coverage, and dimension. Among the various algorithms of Kanonizo, Random exhibits worse results in execution time, real-world, and in the coverage scenarios. Random Search was the best performing algorithm, closely followed by Additional Greedy, whose performance degrades when focusing on execution time.

Up until this point the results reported in this paper represent the average of the set of 11 evaluated projects. To perform a more detailed study, Figures 5a and 5b present the scores

Authorized licensed use limited to: b-on: UNIVERSIDADE NOVA DE LISBOA. Downloaded on February 05,2023 at 14:32:34 UTC from IEEE Xplore. Restrictions apply.

TABLE II: Weights used for MCDM in each scenario.

	Sub-criterion					
Scenario	**File size**	**# of test cases**	**Branches covered**	**Total coverage**	**Mutation score**	**Execution Time**
Real-world	0.0095	0.0473	0.0245	0.2132	0.0569	0.6486
Dimension	0.5727	0.2455	0.0303	0.0303	0.0303	0.0909
Exec. Time.	0.0455	0.0455	0.0303	0.0303	0.0303	0.8182
Mutation Cov.	0.0455	0.0455	0.0744	0.0744	0.6694	0.0909
Total Cov.	0.0455	0.0455	0.0744	0.6694	0.0744	0.0909

obtained on the real-world scenario by all tools for each of the 11 projects, for the Normal and EvoSuite test scenarios. The same cutoff of 30% was used for all algorithms of Kanonizo.

Considering the results for the normal test scenario (Fig. 5a) it is possible to find one project where all tools had very similar performance (`commons-emails`) and various projects where most tools except one had similar performance. This was the case of the `commons-lang` project, where Testler achieved very low scores, and `crunch-core`, `crunch-kafka`, and `xmlsec` where Randoop also achieves low scores. To an extent the same observation can be made regarding the various algorithms of Kanonizo. For example, Greedy performed much worse than the remaining algorithms in the `tudu-lists` project, but Random Search was the best overall tool for the `pmd-core` project and Random was the best in the `lambdaj` project.

When looking at the EvoSuite runs, the picture changes drastically, and projects that once showed good and equilibrated performance across all tools, such as `commons-email`, now have some of the most variable results. The EvoSuite tool generally performed very well but also obtains lower scores in certain projects, such as `crunch-core`, where it had the lowest score from all tools.

In summary, it seems that although some tools, such as EvoSuite, Kanonizo Random Search and Kanonizo Additional Greedy, tend to perform very well for the majority of projects, the scores are still very much dependent on the project being evaluated. Furthermore, the comparison between Normal runs and EvoSuite runs, suggests that one important factor for the performance of a tool is not the project itself, but the test suite being reduced, as big discrepancies were noted for the same project and tool across these two types of runs.

Figures 6a and 6b represent the same information but facilitating the analysis of how each project affects a certain tool's performance.

The research question of whether there is a best TSR tool (relative to others) can be answered positively, as in so far that it was possible to obtain a limited set of tools (EvoSuite, Kanonizo Random Search #30 and Kanonizo Additional Greedy #30) that consistently showed better performance for the majority of projects, but with limitations. Despite the affirmative answer, it should be noted that in order to find the absolute best tool, the interested practitioner *should execute TESRAC using the specific project and test suite that will be reduced*. Only then it will be possible to identify the most adequate tool.

B. RQ2: Can a TCP tool be adapted to effectively be an alternative to a TSR tool?

The previously shown results, namely in Figures 3a, 3b, 4a and 4b, demonstrate that a TCP tool (in this case, Kanonizo) can have performance that is better than some TSR tools (e.g., Testler and Randoop) and closely matches even the tool that the experimental evaluation deemed as the best (EvoSuite). Therefore, this research question can be answered affirmatively. *Yes, it is possible to adapt a TCP tool to be an alternative to a TSR tool* and it can even match the performance of the top-performing TSR tools. In our opinion, this is an interesting observation that might lead to additional research in the future.

C. RQ3: The complexity of the projects influences the reduction score of any tool?

Figures 5 and 6 suggested that the performance of the different tools is sensitive to the project or the test suit being reduced. To further analyze this question, the Pearson correlation coefficient between reduction scores for various scenarios and three project metrics – lines of code (LoC), number of classes, and number of test classes – are presented in Figure 7, which refers only to the normal test scenario. Darker colors mean higher correlation between a reduction score and a metric. A high positive or negative Pearson correlation coefficient means that a variable X (e.g., a project property) is linearly correlated to another variable Y (e.g., the reduction score in a scenario). A coefficient near 0 means that there is no linear correlation.

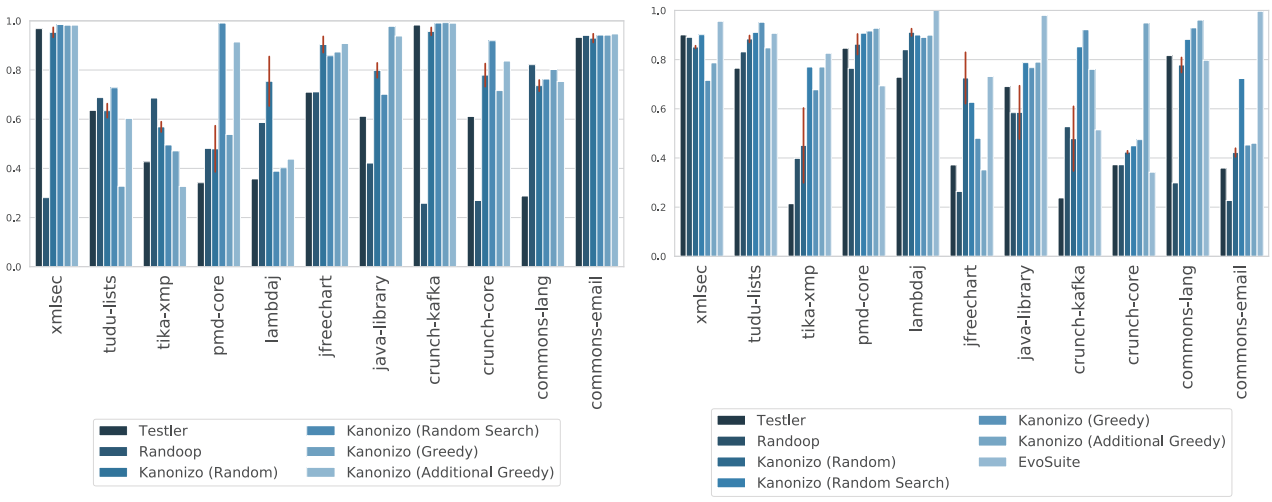
If project properties had a strong impact on the reduction score, then we would expect to see high correlation coefficients, however the heat-map shows only values lower than 0.21. This suggests that there is just a small linear correlation between the number of test classes and the reduction scores on the real world and execution time focused scenarios.

To answer this research question, *no, the complexity of the projects does not influence the reduction score*. However, the results suggest that tools can have varying performance depending on project or test suite, and this observation warrants further research in the future, namely by taking into consideration other properties of the projects such as the type of software in the project.

VI. Discussion and Limitations

The presented results suggest one important conclusion: test case prioritization can be a viable alternative to the TSR tools available, considering Java unit testing. In fact, according to

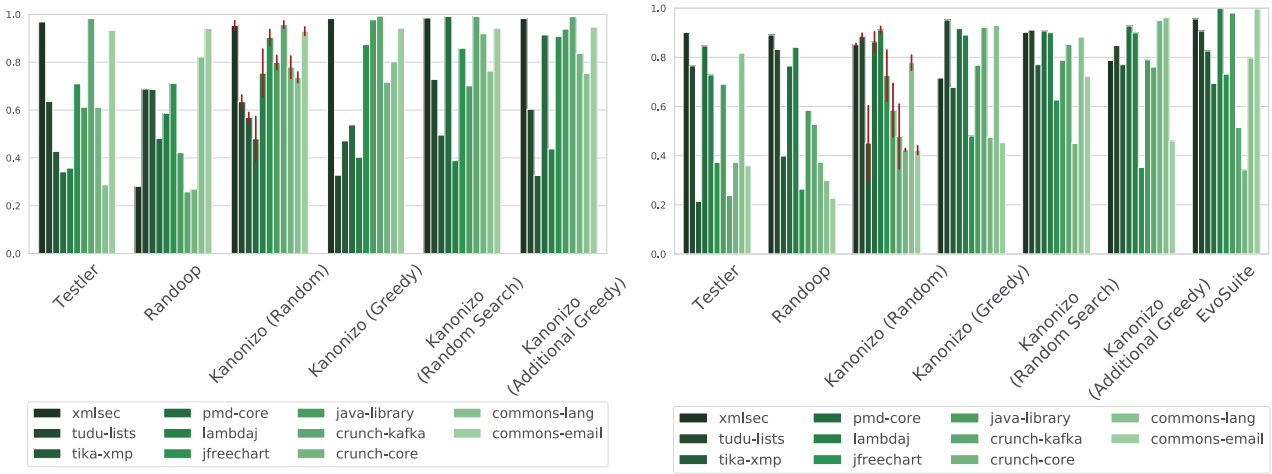
Authorized licensed use limited to: b-on: UNIVERSIDADE NOVA DE LISBOA. Downloaded on February 05,2023 at 14:32:34 UTC from IEEE Xplore. Restrictions apply.



(a) Normal runs.

(b) EvoSuite runs.

Fig. 5: Reduction scores per project and tool on the real world scenario



(a) Normal runs.

(b) EvoSuite runs.

Fig. 6: Reduction scores per tool and project on the real world scenario

Fig. 7: Heatmap with Pearson correlation between scores and project properties.

our own field study, from all the TSR tools that have been published in literature only a few are actively maintained. One possible explanation is that research is more active in the field of TCP and researchers seem to believe that this a more promising area for significant contributions. The other possible explanation is that there is room for new and improved open-source tools that adopt innovative techniques that can advance the state-of-the-art in TSR, such as what Kanonizo and EvoSuite are doing for test case prioritization and test suite generation, respectively. Our framework is by itself a contribution in this area for creating a healthy competition between existing and new TSR tools.

One unexpected result is that EvoSuite (when instrumented) can be a viable TSR tool. This is surprising since its main

Authorized licensed use limited to: b-on: UNIVERSIDADE NOVA DE LISBOA. Downloaded on February 05,2023 at 14:32:34 UTC from IEEE Xplore. Restrictions apply.

goal is not test suite reduction. Our findings are controversial, since one can argue that EvoSuite generates the test suites that it is then going to reduce, giving it an unfair advantage over the remaining tools. However, we defend that, for the projects used in our experimental work, TSR tools should have been more effective than a tool that was instrumented to access its internal test suite reduction mechanism. As it is clear from Figure 4b, EvoSuite generates test suites that take negligibly more time to execute, but have greater code coverage and mutation coverage and have a smaller dimension. It would be an interesting contribution to the community if EvoSuite developers provided direct access to its TSR mechanism (e.g. as a supported API), or as a standalone project that could compete with other TSR tools.

The test case prioritization tool, Kanonizo, was evaluated using four algorithms supported by it, using different cutoff values. The results showed that the best performing algorithms were Random Search, followed by Additional Greedy. The usage of the Random algorithm is discouraged due to its non-deterministic nature and associated probability for failing to find acceptable reduced test suites. In terms of cutoff values, the results suggest that they depend on the test suite (the ideal cutoff value differed between Normal and EvoSuite runs), but generally, cutoff values equal or above 20% should be used.

The relative performance between the evaluated tools tended to remain quite unaffected by the project to be reduced (i.e., if a tool shows good performance in a project, it is likely to perform well in almost all other projects). However, occasionally a tool ranked a lot lower than usual in a specific combination of project and test suite (e.g., Normal and EvoSuite runs). This suggests that there can be properties of a test suite (or project) that may impact significantly the performance of the tool. On the other hand, an analysis of the correlation between a set of project properties (LoC, number of classes, and number of test classes) and the reduction scores concluded that there is little correlation between these two groups.

A threat present in our work is the strong association with the Java language and JUnit, which represent a subset of all programming languages and unit testing frameworks. However, this was a conscious decision that we took in order to define a common platform for testing and promote a fair comparison between the test subjects. Also, Java-based TSR tools are, to the best of our knowledge, receiving an increasing interest from the research community [5], [6]. It is worth mentioning that we devised TESRAC's architecture to be generic enough to be applied in the development of similar assessment tools for other programming languages.

We argue that the sub-criteria (i.e. metrics) used for defining the MCDM quality model (see Figure 2), are adequate to assess and compare the effectiveness of TSR. Nonetheless, the quality model can be easily extended to include other metrics such as requirements coverage, or Average Percentage of Faults Detected (APFD), respectively useful for companies with a requirement-based development process and for practitioners or researchers in the field of fault localization.

The eleven projects that were studied in the evaluation may not be representative of every existing project, however they are popular open-source projects that maintain an adequate set of test suites. In our choice of projects, we also tried to include most of the projects that were tested in previous studies, such as those that were originally used to evaluate Testler [8].

VII. Conclusion

This paper presented TESRAC, a framework for assessing and comparing the effectiveness of test suite reduction tools. Its goal is to foster innovation by encouraging competitiveness between existing and new tools and to raise awareness of the need for new approaches and tools for test suite reduction. We used TESRAC to evaluate the test suite reduction performance over 11 open-source projects of four tools: Randoop, Testler, Evosuite and Kanonizo. We concluded that EvoSuite and Kanonizo provide good temporal and size reduction without sacrificing code and mutation coverage. We also concluded that the performance of any tool, despite being generally consistent, may vary depending on the project and test suite being reduced. As future work we intend to port new test suite reduction techniques from other tools to TESRAC. We also intend to implement support for other code analysis tools, besides OpenClover, and give the user the option to choose which one to use. This could also make for an interesting comparative study about the way the analysis of each one of these tools influence the final score obtained for a TSR tool. Finally, we will consider the implementation of other multi-criteria decision making techniques and study additional metrics for inclusion, as to give more choice to how users compare the effectiveness of existing tools.

Acknowledgment

This work was partially supported by projects NOVA LINCS (FC&T grant UIDB/04516/2020) and NG-STORAGE (FC&T grant PTDC/CCI-INF/32038/2017) and within the scope of project CISUC - UID/CEC/00326/2020, by the European Social Fund, through the Regional Operational Program Centro 2020.

References

[1] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2013.

[2] J. A. P. Lima and S. R. Vergilio, "Test case prioritization in continuous integration environments: A systematic mapping study," *Information and Software Technology*, vol. 121, p. 106268, 2020.

[3] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi, "Regression test selection for java software," *SIGPLAN Not.*, vol. 36, no. 11, p. 312–326, Oct. 2001.

[4] G. Rothermel, R. H. Untch, Chengyun Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 10, pp. 929–948, 2001.

[5] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[6] S. U. R. Khan, S. P. Lee, R. W. Ahmad, A. Akhunzada, and V. Chang, "A survey on test suite reduction frameworks and tools," *International Journal of Information Management*, vol. 36, no. 6, Part A, pp. 963 – 975, 2016.

Authorized licensed use limited to: b-on: UNIVERSIDADE NOVA DE LISBOA. Downloaded on February 05,2023 at 14:32:34 UTC from IEEE Xplore. Restrictions apply.

[7] G. Fraser and A. Arcuri, "Evosuite: Automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. NY, USA: ACM, 2011, pp. 416–419.
[8] A. Vahabzadeh, A. Stocco, and A. Mesbah, "Fine-grained test minimization," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. NY, USA: ACM, 2018, pp. 210–221.
[9] C. Pacheco and M. 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*, ser. OOPSLA '07. NY, USA: ACM, 2007, p. 815–816.
[10] D. Paterson and J. Campos, "Kanonizo," https://github.com/kanonizo/kanonizo, 2020, accessed: 2020-08-01.
[11] J. R. Horgan and S. London, "A data flow coverage testing tool for c," in *Proceedings of the Second Symposium on Assessment of Quality Software Development Tools*, May 1992, pp. 2–10.
[12] J. H. Andrews, S. Haldar, Y. Lei, and F. C. H. Li, "Tool support for randomized unit testing," ser. RT '06. NY, USA: ACM, 2006, p. 36–45.
[13] J. M. Kauffman and G. M. Kapfhammer, "A framework to support research in and encourage industrial adoption of regression testing techniques," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, April 2012, pp. 907–908.
[14] L. Zhang, J. Zhou, D. Hao, L. Zhang, and H. Mei, "Jtop: Managing junit test cases in absence of coverage information," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '09. USA: IEEE, 2009, p. 677–679.
[15] F. Dadeau, Y. Ledru, and L. Du Bousquet, "Directed random reduction of combinatorial test suites," ser. RT '07. NY, USA: ACM, 2007, p. 18–25.
[16] S. Wang, S. Ali, and A. Gotlieb, "Cost-effective test suite minimization in product lines using search techniques," *Journal of Systems and Software*, vol. 103, pp. 370 – 391, 2015.
[17] OpenClover, "OpenClover," https://openclover.org/, 2020, accessed: 2020-08-01.
[18] H. Coles, "PIT Mutation Testing," https://pitest.org/, 2020, accessed: 2020-08-01.
[19] M. Martínez, D. D. Andrés, and J. Ruiz, "Gaining confidence on dependability benchmarks' conclusions through "back-to-back" testing (practical experience report)," in *2014 Tenth European Dependable Computing Conference*, 2014, pp. 130–137.
[20] OpenClover, "OpenClover code metrics," https://openclover.org/doc/manual/4.2.0/general--about-openclover-code-metrics.html, 2020, accessed: 2022-02-01.
[21] T. L. Saaty, "Decision making with the analytic hierarchy process," *International journal of services sciences*, vol. 1, no. 1, pp. 83–98, 2008.

Authorized licensed use limited to: b-on: UNIVERSIDADE NOVA DE LISBOA. Downloaded on February 05,2023 at 14:32:34 UTC from IEEE Xplore. Restrictions apply.