

© 2017 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.

Alessandro Marchetto, Giuseppe Scanniello, and Angelo Susi, **Combining Code and Requirements Coverage with Execution Cost for Test Suite Reduction**, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, Volume: 45, Issue: 4, April 1 2019, DOI: 10.1109/TSE.2017.2777831

The final published version is available online at: <https://ieeexplore.ieee.org/document/8120000>

When citing, please refer to the published version.

Combining Code and Requirements Coverage with Execution Cost for Test Suite Reduction

Alessandro Marchetto, Giuseppe Scanniello, *Member, IEEE*, and Angelo Susi, *Member, IEEE*

Abstract—Test suites tend to become large and complex after software evolution iterations, thus increasing effort and cost to execute regression testing. In this context, test suite reduction approaches could be applied to identify subsets of original test suites that preserve the capability of satisfying testing requirements and revealing faults. In this paper, we propose Multi-Objective test suites REDuction (named *MORE+*): a three-dimension approach for test suite reduction. The first dimension is the structural one and concerns the information on how test cases in a suite exercise the under-test application. The second dimension is functional and concerns how test cases exercise business application requirements. The third dimension is the cost and concerns the time to execute test cases. We define *MORE+* as a multi-objective approach that reduces test suites so maximizing their capability in revealing faults according to the three considered dimensions. We have compared *MORE+* with seven baseline approaches on 20 Java applications. Results showed, in particular, the effectiveness of *MORE+* in reducing test suites with respect to these baselines, i.e., significantly more faults are revealed with test suites reduced by applying *MORE+*.

Index Terms—Multi-objective approach, Regression Testing; Testing; Test Suite Reduction.



1 INTRODUCTION

REGRESSION testing is conducted after software evolution operations, e.g., enhancements and patching. Regression testing aims to: (i) guarantee that software evolution operations do not compromise the expected behavior of a software application and (ii) optimize an original/existing test suite that tends to grow larger and to become complex after the execution of these operations. During regression testing, a software engineer conducts several technical and business activities that affect the success or failure of a software project [1]. Relevant activities are: (i) test case prioritization; (ii) test case selection; and (iii) test suite reduction [2], [3], [4]. Test case prioritization concerns the identification of the ordering of test cases that maximizes desirable properties, such as early fault detection. Test prioritization aims to early detect the presence of faults but, sometimes, it has been also combined with diagnosis to speed-up the faults localization, e.g., [5]. Test case selection approaches seek to identify test cases that are relevant to test recent changes in an application. Test suite reduction approaches seek to “minimize” the original test suite by reducing the number of tests to be executed for testing subsequent versions of the application (e.g., removing redundant test cases) and, at the same time, to preserve test requirements of the original test suite. Test suite reduction is sometimes also called minimization, meaning that the elimination of test cases from the original test suite is not permanent [2].

Although test case selection and test suite reduction are often (wrongly) used interchangeably, they are different [2]. One of the most remarkable difference is that test suite reduction approaches reduce test suite by preserving the test requirements of the original suite in the reduced one. Test suite reduction approaches often use information about the last-tested application version (e.g., code coverage, fault locations and density) to identify test cases to be removed (e.g., redundant test cases), thus preserving those test cases that exercise the changed parts of a subsequent version of an application. Instead, test case selection: (i) is temporary and specific for a pair of application versions (e.g., the last-tested application version and the subsequent to-be-tested one); (ii) does not aim to preserve the test requirements of the original test suite, but it focuses on the selection of test cases that test the changed part of the application source code; and (iii) often uses *diff* operations between the code of last-tested and new to-be-tested application versions to identify changes in the new to-be-tested application [2], [6].

A number of approaches for test suite reduction has been proposed in the literature [2], [7], [8], [9], [10], [11], [12]. To evaluate them and compare one another a number of empirical studies have been also conducted [9], [10], [13], [14]. However, these studies achieved contrasting results.

For instance, Rothermel et al. [13] reported that fault-detection capability of reduced test suites can be sensibly worse than original test suites. Conversely, Wong et al. [14] and Zhang et al. [9] showed that reduction approaches could produce test suites quite competitive with respect to the original ones. Existing approaches (e.g., [7], [8], [9], [10], [11]) are mostly based on a single dimension and exploit code coverage as a proxy for estimating the capability of a suite in detecting faults. Instead, only a few attempts exist to reduce test suites on the basis of multiple dimensions [2]. These approaches [2], [7],

- A. Marchetto is an Independent researcher.
E-mail: alex.marchetto@gmail.com
- G. Scanniello is with DiMIE - University of Basilicata.
E-mail: giuseppe.scanniello@unibas.it
- A. Susi is with Fondazione Bruno Kessler - Trento, Italy.
E-mail: susi@fbk.eu

[12] mainly consider *structural* information (e.g., code coverage) and *cost* (e.g., time) to execute test cases and ignore *functional* information (e.g., business requirements¹ coverage). However, recent results show that: (i) the use of code coverage as proxy measure for estimating the capability in finding faults of a test suite is not always adequate as expected [15] and (ii) considering multiple dimensions is quite promising [16], [17], [18], [19], [20]. In our view, these results suggest that explicitly considering low-level information (e.g., structural information about the application such as code coverage and cost) and high-level information (e.g., coverage of functional information such as the ones described in application requirements) could be promising to overcome the results achieved by traditional test suite reduction approaches. To preliminarily investigate on this view, we have proposed *MORE* (Multi-Objective test suite REduction) [21]. It uses the Non-dominated Sorting Genetic Algorithm-II (NSGA-II) [22] to reduce a test suite by applying a three-dimensional analysis of test cases: (i) *structural* dimension concerns information regarding how test cases exercise the source code of the application under test, (ii) *functional* dimension concerns coverage of application requirements, and (iii) *cost* dimension is about the time to execute test cases. To establish a relationship among these three dimensions, *MORE* exploits Latent Semantic Indexing (LSI) [23] to compute lexical similarity among application code, test cases, and application requirements. LSI is an indexing and retrieval method that discovers existing patterns in the relationship between terms and concepts contained in textual documents. The underlying concept of LSI is that terms that are used in the same context have similar meanings. We have also reported in [21] a proof of concept in which *MORE* was applied to three Java applications and compared it with three baseline approaches. Obtained results were promising.

In this paper, we present and evaluate an extended version of *MORE*, we named it *MORE+*. Several are the differences between *MORE* and *MORE+*. The most important one is that *MORE+* considers the fault-proneness of different parts of both application code and application requirements, when reducing test suites. In other words, *MORE+* reduces test suite by focusing on those test cases that better test fault-prone parts of the application under testing. Fault-prone parts of the application are automatically identified in *MORE+* by applying some maintainability indexes. To assess whether test suites reduced by *MORE+* may be effective as whole test suites, we have conducted a large experimental evaluation on 20 Java applications (*MORE* was originally evaluated on only three small Java applications). We have also compared *MORE+* with seven baseline approaches. The trends observed in the results suggest that *MORE+* outperforms existing approaches since it finds more faults, at the cost of more time for its execution. For example, *MORE+* outperforms existing reduction approaches in preserving a high capability to dis-

cover faults. In particular, it is able to find in between 5.9% and 13.4% (with an average of 10.2%) more faults than baseline approaches. A possible drawback of *MORE+* is that it might be costly since it computes lexical similarity and we observed that this operation requires on average 2.7 minutes, but it ranges from 0.2 seconds to 14 minutes, depending on the considered application. Although our approach tends to be costly with respect to others, it is able to find more faults than them, thus we observed that it is cost-effective with respect to the existing approaches even if it is applied to one version of the application to be tested. This might be acceptable in many regressions testing scenarios, e.g., in all those cases where a reduced test suite is frequently reused to test several versions of a given application.

We can summarize the main contributions provided in this new paper as follows:

- *MORE+* uses NSGA-II [22] and adopts a binary-vector encoding, rather than a permutation of a fixed size as well as done by the original version of the algorithm, and a set of genetic operators for better exploring the solution space. Thanks to the new algorithm, *MORE+* could also provide the reduced test suites without the need of pre-specifying the size of the reduced test suite (e.g., 30% of the size of the original test suite), as well as requested instead by our original *MORE*.
- The approach adopted by *MORE+* performs a three-dimensional analysis of test cases and additionally exploits software metrics to estimate the most fault-proneness parts of the application source code and requirements. By means of software metrics, *MORE+* weights application code and requirements aiming to preserve test cases that better exercise fault-prone parts of the application.
- An extensive evaluation of *MORE+* through a large empirical study with 20 Java applications, seven baseline approaches for comparison, and a number of criteria representing the standard in the literature for the assessment of test suite reduction approaches.

Paper structure. Related work and background are highlighted in Section 2, while *MORE+* is presented in Section 3. The design of our empirical study and possible validity threats are shown in Section 4. The obtained results are reported in Section 5, additional analysis documented in Section 6, while outcomes and their practical implication are discussed in Section 7. Final remarks conclude the paper in Section 8.

2 RELATED WORK AND BACKGROUND

A number of approaches have been suggested to aid regression testing. The three major branches include test case prioritization, test case selection, and test suite reduction. As mentioned before, these branches are different one another because they address different concerns related to regression testing. Since we are proposing a test suite reduction approach, we focus the discussion on approaches

1. The term business (application) requirements indicates all requirements that are driven by business needs, e.g., product and process requirements.

conceived for the reduction/minimization of test suites. In the survey by Yoo and Harman [2], the interested reader can find information on test case prioritization and on test case selection approaches. We conclude this section by presenting background information on LSI and research work on the recovery of traceability links.

2.1 Traditional Approaches

As for test suite reduction, a number of techniques and approaches have been proposed [2]. Most of the traditional approaches use heuristic-based criteria to identify and discard redundant test cases. If a test case satisfies a subset of the test specifications of another test case, it could be considered a redundant test case. For instance, Chen and Lau [24] proposed heuristics to identify essential test cases of a suite, those test cases minimizing the number of unsatisfied test specifications. Other traditional approaches reduce test suites considering code and data-flow coverage information (e.g., lines of code, statements, definition-use associations) [7], [8], [9], [10], [11]. For example, Campos and Abreu [11] encoded existing relationships between test cases and testing requirement (i.e., code coverage) in a coverage matrix. This information was then used to derive a set of constraints and compute a collection of optimal minimal sets (preserving code coverage capability of original suites as much as possible). Traditional approaches tend to focus on one aspect (e.g., code coverage) to reduce test suite. Malishevsky et al. [25] introduced the notion of cost-cognizant for test case prioritization, other researchers introduced and evaluated traditional test suite reduction approaches for allowing them to explicitly consider two aspects (e.g., code coverage and execution cost) at the same time. For instance, Smith et al. [8], [26] empirically compared four traditional test suite reduction approaches that consider code coverage to reduce test suite and extended them for considering also test cost. In particular, they consider: (i) traditional greedy approach (GRD) that reduces test suites by identifying redundant test cases according to the relationships among test cases in terms of code coverage; (ii) Harrold Gupta Soffa (HGS); and (iii) the Delayed greedy (DGR) approaches that represent two variants of the greedy approach that perform additional analysis of the coverage information before making any greedy choice; and (iv) the 2-optimal greedy (2OPT) that is another variant that conduces all-pairs comparison of test cases. Their results show that by incorporating test case cost (two-objective) in the suite optimization, other than code coverage, the greedy-based approaches outperform their traditional (single-objective) implementations.

Other approaches take into account test execution profiles (e.g., [12]) or focus on execution costs to reduce test suites (e.g., [27]). These approaches execute test cases to collect information on the code they exercise [12] or on the execution cost [27]. This information is then used to profile test cases and identify the ones that largely exercise the application [12] or that lower the cost of the reduced suites [27].

Few approaches exist that reduce test suites considering how test cases exercise application requirements. For instance, Selvakumar and Ramaraj [28] proposed a specification based approach. They assumed that each change in the specifications raises the need of regression testing. To reduce test suites, authors adopted a specification-dependency analysis generated from specifications. Later, Gotlieb and Marijan [29] suggested a test suite reduction approach that considers connection with the coverage of requirements. Specifically, a flow network is formed by using a given test suite and requirements this suite covers. Then the Ford-Fulkerson method and Constraint Programming techniques are applied to compute maximum flows and to search for optimal flows.

Recently, Shi et al. [30] showed that even if test selection and test suite reduction are implicitly different they can be potentially combined. Indeed, authors suggested that if there is a need to speed up testing, then combining both test suite reduction and test selection is worthwhile. This combination can save in the number of tests as long as one a loss in fault-detection capability is not a major issue for the tester.

With respect to the approaches highlighted, *MORE+* instances an evolutionary algorithm to balance among three different objectives when reducing test suites. This allows us to better explore the solution space by equally considering these three objectives to choose test cases when reducing test suites.

A recent trend related to regression testing concerns the reduction of individual test cases instead of entire test suites. Reducing a test case means to identify its atomic parts and remove some of them, according to an adequacy criterion. For example, typically a unit test case is composed of a sequence of function calls and then test case reduction approaches remove some of these function calls according to some criteria. Test case reduction approaches, e.g., Groce et al. [31] and Alipour et al. [32], aim to reduce a test case while trying to preserve its fault-detection capability. Groce et al. [31] proposed cause reduction, an approach to reduce test cases by preserving the coverage capability of the original test cases, namely reduced test cases completely cover all the code elements originally covered by the test cases. Alipour et al. [32] proposed a test-case reduction approach that allows them to only partially preserve a property (e.g., code coverage). That is, reduced test cases partially cover code elements originally covered by the test cases.

2.2 Multi-objective Approaches

The largest part of the approaches for test suite reduction is single-objective (e.g., [2], [7]). However, multi-objective techniques have been proposed. For instance, Mirarab et al. [33] presented an approach to reduce a test suite by selecting a predefined number of test cases. The approach codified test case selection as an Integer Linear Programming problem by applying a function based on two criteria (code coverage and coverage of changed code) to

get final solutions. Differently, Shi et al. [17] presented an approach that reduces test suites by considering two objectives: code coverage and number of killed mutants. They reduced test suites by considering: (i) how test cases cover source code and (ii) their capability to detect automatically injected faults. In most cases, evolutionary algorithms were exploited by formulating test suite reduction as an optimization problem [16], [18], [19]. An evolutionary algorithm is a population-based metaheuristic optimization algorithm [34], [35] that uses mechanisms inspired by biological evolution, such as reproduction, mutation, recombination, and selection for discovering (sub-)optimal solutions of a problem within a reasonable time. In specific evolutionary algorithms, such as in the 1+1 evolution strategy, the population could be composed of 1 or few elements. Generally speaking, an evolutionary algorithm manipulates population of solutions for generating such optimal solutions. The algorithm iteratively evolves an initial population, that is a set of candidate solutions of the problem under consideration, for searching over the solution space of the problem. The evolution of the population is conducted by applying specific genetic operators (e.g., reproduction, mutation) for changing the population and a fitness function that has to be customized for determining the quality of the solutions. In the population evolution, solutions with higher fitness values are more likely to be preserved and evolved. Often, the algorithm iterations terminate when either a maximum number of generations have been produced, or a satisfactory fitness level has been reached for the population. Evolutionary algorithms are broadly applicable and easily tailored to specific problems. Existing approaches for test suite reduction based on an evolutionary algorithm often consider either code or requirement coverage information and try balancing that information with execution cost of test cases as follows: (i) explicitly optimizing them as two objectives (e.g., code coverage and execution cost) and (ii) redefining a multi-objective optimization problem to a single-objective by using optimization functions conflating more objectives into one. For instance, Yoo and Harman [16] showed benefits of Pareto optimality for test case selection and test suite reduction (i.e., test suite minimization because the elimination of test case is not permanent), respectively. The authors presented a two-objective approach in which code coverage and execution cost are explicitly considered. To reduce test suites, MA et al. [18] adopted an objective function that conflates code coverage and execution cost information. Furthermore, de Souza et al. [19] exploited the Particle Swarm Optimization (PSO) algorithm and considered two objectives: requirements coverage and test case execution cost.

A recent study by Costa [36] proposed a three-objective approach that reduces test suites and, at the same time, tries to produce suites that maximize their capability of locate faults. To this aim, they consider code coverage, execution time and ϕ , a metric related to the capability of a test suite to locate faults by adopting autonomic fault localization techniques. This work presents a number of differences with

respect to *MORE+*. The most remarkable difference is that our *MORE+* focuses on application requirements to reduce test suites.

Our approach is inspired to that presented by Yoo and Harman in [16]. The most remarkable difference is that they reduced test suite by focusing on low-level information, such as code coverage and test case execution cost. Conversely, we propose an approach to reduce test suites by considering both low- (i.e., code coverage and execution cost) and high- level information (i.e., application requirements coverage) of each test case. We fill the gap between these kinds of information by using LSI [23].

Although test suite reduction and test case prioritization are two formally different problems, they can be considered related one another [2]. In a previous work, we have investigated the possibility of considering test case prioritization as a multi-objective optimization problem [3], [37]. We aimed to determine the ordering of test cases that maximize the number of detected faults that are both technical (implementation) and business critical (application requirements). To this end, we adopted a customized version of the Non-dominated Sorting Genetic Algorithm-II (NSGA-II) algorithm and used code and requirements coverage information as well as the cost to execute test cases. To increase the body of knowledge on the usefulness of structural, functional, and cost dimensions in software regression testing, we decided to use our gained experience to define a new approach (i.e., the one presented in this paper) for test suite reduction. Outcomes further supported our initial hypotheses (i.e., technical and business dimensions are both useful in regression testing) and advanced the state-of-the-art in exploiting both low- and high- level information to test suite reduction.

2.3 Empirical Investigations in the Context of Test Suite Reduction

To evaluate costs and benefits of test suite reduction techniques and approaches, a number of empirical studies has been conducted [9], [10], [13], [14]. For example, Zhong et al. [10] showed that different reduction approaches produced different test suites even if these approaches were applied in the same context and reduction degree of test suites was the same. The experiments documented in the literature achieved contrasting results. For instance, Rothermel et al. [13] reported that fault-detection capability of reduced test suites can be sensibly worse than whole test suites. Conversely, Wong et al. [14] showed that representative sets of test cases had almost the same capability to reveal faults as original test suites. Zhang et al. [9] showed that traditional reduction approaches could be quite competitive in reducing size of test suites, reasonably preserving their capability in detecting faults. Contrasting results suggest further empirical investigations on this matter. This is also why we conducted an extensively evaluation of *MORE+* comparing it with a number of baseline approaches for test suite reduction [8], [9], [10], [16], [26].

```

Examjava
7 public class Exam implements Serializable {
8     public String name;
9     public int cfu;
10    public int vote;
11    public boolean laude;
12    public boolean maked;
13
14    public Exam(){
15        name = "Unknown";
16        cfu = -1; vote = -1;
17        laude = false; maked = false;
18    }
19
20    public static Exam getInstance(String name, String cfu,
21        String vote){
22        Exam e = new Exam();
23        e.setName(name);
24        e.setCfu(Integer.parseInt(cfu));
25        if (vote.trim().endsWith("laude")) {
26            e.setVote(30);
27            e.setLaude(true);
28            e.setMaked(true);
29        } else if (vote.indexOf("-") == -1){
30            e.setVote(Integer.parseInt(vote));
31            e.setMaked(true);
32        } else e.setMaked(false);
33        return e;
34    }
}

AddExam.txt
1 A student can add a new exam to the register.
2 An exam is composed of a name, CFU
3 (i.e., a number that represent the university credit
4 of the exam) and an optional vote.
5 The name is unique, CFU is a positive number (>=0)
6 and the vote, if inserted, is a number included
7 between 0 and 30 (the vote can be also 0 or 30).
8 A vote < 18 is negative (i.e., the exam is not passed)
9 while >= 18 is positive (i.e., the exam is passed).
10 An exam can be inserted also without the vote;
11 it can be inserted later. 'Laude' can be added
12 only when the vote is 30.
13

```

Fig. 1. Example of a Java class (top) and application requirement (bottom)

2.4 LSI and Traceability/Similarity Links

A traceability link is an association between two software artifacts that represents the existence of a relationship (e.g., overlap, dependency, contribution, evolution, refinement, or conflict) between such artifacts. Traceability links among software artifacts provide important insights into the phases of design, development, evolution, and testing [38], thus allowing software engineers to understand relationships that exist within and across different kinds of software artifacts (e.g., requirements specifications and code). Traceability links are very often not well documented and aligned with software implementation. Therefore, automated techniques and tools might be needed to infer candidates of traceability links among software artifacts of different kinds (e.g., [39], [40], [41], [42], [43], [44]). Figure 1 shows an example of two textual-based software artifacts associated one another: (top) a Java class and (bottom) an application requirement.

Basically, approaches for the recovery of traceability links compute the similarity among software artifacts. Information retrieval (IR) techniques have been suggested to compute the lexical similarity among textual-based representations of software artifacts. For instance, among the most well-know and used techniques we can find: Latent Semantic Indexing (LSI) [23], Vector Space Model (VSM) [45] and Latent Dirichlet Allocation (LDA) [46].

These IR-based techniques, have been used to recovery links among software artifacts such as application code and test cases, application requirements and test specifications, application code and requirements. Before applying IR-based techniques, software artifacts have to be preliminary analyzed to prepare the corpus to be analyzed. The pre-

liminary analysis mainly consists in: (i) defining textual representations of each software artifact to analyze; and (ii) normalizing such textual representations by splitting words, removing stop words and applying word stemming techniques [45]. This allows the creation of textual-based documentation of software artifacts that have to be compared by means of IR techniques.

Existing research is contradictory about which text retrieval model and technique work best on source code. For example, Marcus and Maletic [43] observed that LSI performs at least as well as VSM [45] and in some cases LSI outperforms VSM. Conversely, Abadi *et al.* [47] observed that VSM provides better results than LSI. Similar results were also obtained by Wang *et al.* [48]. Instead, other authors advocate for the use of LDA [46].

In this work, we used LSI to compute the textual similarity among different kinds of software artifacts. We opted for LSI because it is efficient and has been widely used in traceability recovery although the contrasting results mentioned just before.

LSI assumes that there is some underlying or *latent structure* in word usage that is partially obscured by variability in word choice. LSI uses statistical techniques to estimate this latent structure. In particular, a Singular Value Decomposition (SVD) is applied to a $m \times n$ matrix C (also named term-by-document matrix), where m is the number of terms and n is the number of documents (artifacts in our case). SVD constructs a low-rank approximation C_k to a term-document matrix, for a value of k (i.e., the dimensionality reduction of the latent structure) that is far smaller than the original rank of C . Thus, each row/column is mapped to a k – dimensional space, which is defined by the k principal eigenvectors (corresponding to the largest eigenvalues) of CC^T and $C^T C$. The matrix C_k is itself still an $m \times n$ matrix, irrespective of k . The choice for a value for k is critical; k should be large enough to fit the real structure in the text, but small enough so that we do not also fit the sampling error.

To increase the performance of LSI, the term-by-document matrix could be weighted by using the normalized *term frequency-inverse document frequency* (tf-idf). This schema allows weighting the relevance of each term with respect to the set of documents under analysis. In particular, tf-idf calculates a value for each term by determining its relative frequency in specific documents compared to the inverse proportion of that term in the whole document set. The core idea is that terms that are related to a topic will appear in a limited number of documents, while common terms will be frequently present in several documents of the corpus. To compute similarities between vectors, often cosine similarity between each pair of source and target software artifacts represented in the k -dimensional space [45]. The larger the cosine similarity value, the greater the similarity between source and target artifacts is. All the possible pairs are reported in a ranked list. Irrelevant pairs of artifacts can be removed using a similarity threshold [50] that allows selecting only a subset of top links, i.e., the retrieved links. For example, we use a

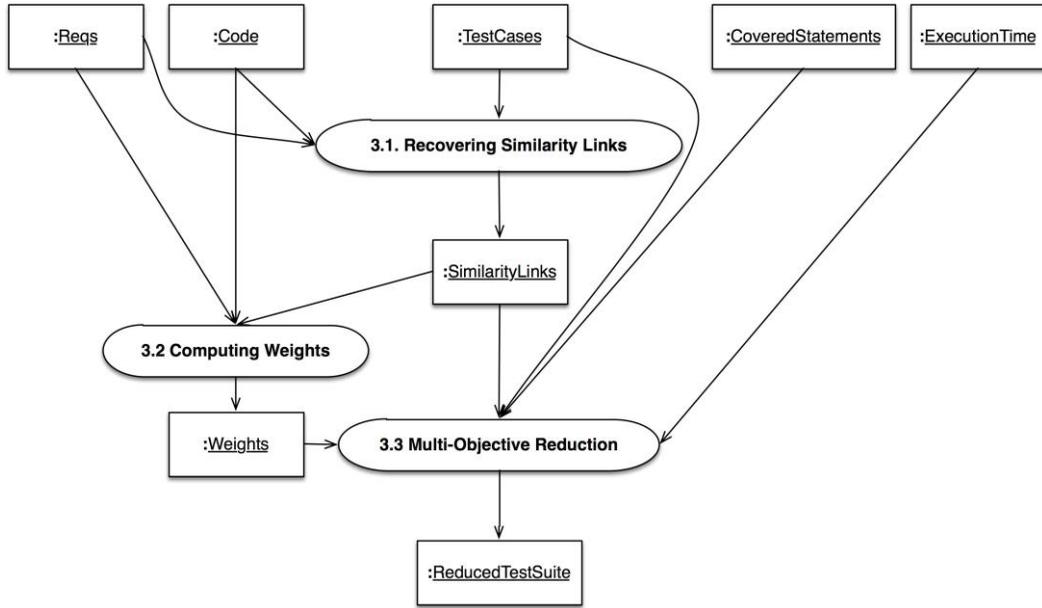


Fig. 2. High-level view of the process implemented in *MORE+* by means of a UML Activity Diagram [49] with object flow. Rectangles are objects, while ellipses are steps/phases of the process.

constant threshold in our approach. Inferring links is clearly a difficult and expansive task. In terms of time complexity, we can consider a complexity of $O(m^2 n+n^3)$ for the SVD computation on a matrix having m and n as dimensions, i.e., terms and documents respectively. We have then to consider an additional linear complexity due to the link inference operations that requires the computation of cosine similarity.

It is worth noting that when inferring a set of candidates traceability links by means of IR techniques, human effort is required to validate these links to obtain actual traceability links. This validation is costly and time consuming (e.g., [39], [41]), as well as it requires high knowledge about the analyzed artifacts. Therefore, recovered candidate links could be used without an actual validation. In such a case, candidate links are simply similarity links.

3 *MORE+*: THE APPROACH

MORE+ exploits a multi-objective evolutionary algorithm (NSGA-II) to reduce test suites according to three objectives: *source code coverage*, *application requirements coverage*, and *cost to execute test cases*. In this way, the evolutionary algorithm aims to maximize the number of detected faults that are both technical (fault at code-level, e.g., implementation and coding faults such as data-flow fault, wrong logic implementation, wrong initializations, wrong code-structure sequence) and business critical (fault at level of application requirements, e.g., missing or improper core-requirements implementation, incorrect input/output of core-requirements, and incorrect core-requirements integration).

- *Input*. Given an application A , *MORE+* requires: (i) its application code; (ii) a set of application requirements $Reqs$ written in natural language; (iii) a test suite S ;

and (iv) information about test case coverage (which code statements are covered by each test case) and execution cost (or also execution time, from here on). We suppose, without losing in generality, that the information on code coverage and execution cost is available from the last tested version of the application. This assumption is the rule for test suite reduction approaches [2].

- *Goal*. *MORE+* aims to produce a reduced test suite S_{red} , a subset of S , by giving high priority to test cases that cover application code and requirements with higher likelihood to be effective in discovering faults. Then, S_{red} , eventually integrated with new test cases, could be devoted to test subsequent versions of A .

- *Approach*. *MORE+* first collects the input information, then it executes the overall process depicted in Figure 2. This process is composed of the following main steps:

- 1) *Recovering Similarity Links*. *MORE+* recovers textual similarity links (i.e., `:SimilarityLinks` in Figure 2) that identify relationships between application requirements (`:Reqs`) and source code (`:Code`) and between application requirements and test cases (`:TestCases`).
- 2) *Computing Weights*. *MORE+* identifies potential fault-prone portions of code (`:Code`) and requirements (`:Reqs`). This is automatically made via a metric-based approach. Software metrics are measured for each code class and application requirement. Similarity links (`:SimilarityLinks`) between code and requirements are used to identify classes implementing each requirement and then used in quality models for estimating the fault-proneness of each code class and requirement. The phase Computing Weights weights (`:Weights`) code classes and requirements according to their estimated fault-proneness.

3) *Multi-Objective Reduction*. *MORE+* relies on the NSGA-II algorithm [22] to identify optimal test suite reductions of S . We opted for NSGA-II because it is widely used for optimization problems in software engineering (e.g., [51]). We used the algorithm to obtain reduced suites by considering the information collected during the test cases execution (covered statements — :CoveredStatements — and execution time — :ExecutionTime) and similarity links recovery (:SimilarityLinks). In this phase, the test cases in the original test suite (:TestCases) are also needed to identify :ReducedTestSuite.

In the following subsections, we detail the main steps of our approach.

3.1 Recovering Similarity Links

MORE+ uses LSI [23] (with a *tf-idf* schema) to recover similarity links (:SimilarityLinks) among software artifacts (:Reqs, :Code, and :TestCases). We opted for LSI because it is efficient and widely used in several text retrieval problems [45] and in the traceability recovery field [43], [50], in particular. The use of a different text retrieval model does not alter our general approach, but it still represents a future direction for our research.

As preprocessing step, we performed textual normalization of software artifacts by removing non-textual tokens, splitting terms composed of two or more words, and eliminating all terms from a stop word list and with a length less than three characters. Finally, a Porter stemmer [45] was applied on the lexemes to reduce them to their root form (e.g., *designing* and *designer* lead to the common radix *design*).

To compute similarities between vectors, we used cosine similarity between each pair of source and target software artifacts represented in the k -dimensional space [45]. All the possible pairs are reported in a ranked list (pairs with a higher similarity values appear first). Irrelevant pairs of artifacts are removed using a similarity threshold [50] that allows selecting only a subset of top pairs. We use a constant threshold (threshold=0.1). We used $k=300$ and 0.1 as constant threshold. These values were chosen on the basis of the outcomes attained in our previous studies [21], [37], [52]

3.2 Computing Weights

MORE+ automatically weights both code w_c and requirements w_r of the application under testing (i.e., :Reqs :Code, respectively). We adopted an approach inspired by the one of Lincke et al. [53] to measure a *Maintainability Index* (MI_{class}) for each class based on a set of code metrics (e.g., Lines Of Code). We use this index as a proxy for estimating the fault-proneness of each class, thus for prioritizing classes. Similarly, *MORE+* computes a Maintainability Index (MI_{req}) for each requirement and it uses this index to prioritize requirements. That is, this index is used to estimate more fault-proneness requirements. These orderings are exploited to select test cases in the test

suite that exercise the most critical (fault-proness) classes and requirements, i.e., top-ranked for MI_{class} and MI_{req} .

The *MORE+* weighting schema also requires as input the similarity links (:SimilarityLinks) between code (i.e., source artifacts) and requirements (i.e., target artifacts). Code and requirements are considered as plain text. Such links are processed by our weighting schema in the following steps: (i) Computing software metrics, (ii) Computing maintainability indexes, and (iii) Prioritizing classes and requirements. We describe these steps in the following subsections.

3.2.1 Computing software metrics

In Table 1, we report the software metrics *MORE+* uses. In particular, we report the name of each metric, the reference to the paper that originally defined it, the software property it measures, and its definition.

To measure MI_{class} for each class (in :Code) of the application under test, we adopt the object-oriented class-level metrics (see Table 1 top) adopted in [53]. Instead, to measure MI_{req} for each application requirement (:Reqs), we adopt two sets of metrics working at different level of granularity: traditional object-oriented metrics working at *class-level* (Table 1 top); and concern-oriented metrics working at *requirements-level* (Table 1 bottom). These two sets of metrics let us measure: (i) each class implementing a requirement in isolation (class-level metrics), and (ii) each group of classes implementing each requirement as-a-whole (concern-oriented metrics).

3.2.2 Computing maintainability indexes

The maintainability index is computed identifying outliers and using quality metrics as in the following steps:

- 1) *Outliers identification*. After computing code and requirements-level metrics, *outliers* have to be identified [53]. Outliers are those elements (i.e., classes or requirements) having metric values within highest/lowest 15% of the value range defined by all elements of application [53]. For instance, if the value of CBO ranges in between 0 and 56. Given two classes having $CBO(c1) = 52$ and $CBO(c2) = 35$, then $c1$ is an outlier for CBO (i.e., the value of $c1$ is in the range 85-100% of CBO), while $c2$ is not an outlier.
- 2) *Software Quality model*. In Table 2, we present our software quality models (inspired to the one presented by Lincke et al. [53]) to compute MI for each class c and requirement r , starting from the metrics reported in Table 1. In the models, metrics are weighted according to the software property they measure. As for class-level and consistently with Lincke, we chose: 2 for coupling, cohesion, and inheritance metrics, while 1 for other metrics. As for requirements-level, we chose: 2 for size and scattering, while 1 for other metrics.
- 3) *Maintainability index computation*. By knowing outliers and using software quality models, we compute

TABLE 1
Metrics for the automatic weighting used in *MORE+*

Metric	Ref.	Property	Definition
Class-level Metrics			
(<i>CBO</i>) Coupling Between Objects	[54]	Coupling	It is the number of classes to which a class is coupled
(<i>RFC</i>) Response For a Class	[54]	Coupling	It is the set of methods that can potentially be executed in response to a message received by an object of the class
(<i>LCOM</i>) Lack Of Cohesion on Methods	[54]	Cohesion	It describes the lack of cohesion among methods of a class
(<i>LOCs</i>) Lines Of Code	-	Size	It counts the lines of code of a class
(<i>NOM</i>) Number of methods	-	Size	It counts the number of methods of a class
(<i>DIT</i>) Depth of Inheritance Tree	[54]	Inheritance	It is the length of the class from the root of the inheritance tree
(<i>NOC</i>) Number of Children	[54]	Complexity	It is the number of immediate subclasses of the class in the class hierarchy
(<i>MCC</i>) McCabe Cyclomatic Complexity	[54]	Complexity	It is (median of) the number of flows thought the code of the method of a class
(<i>WMC</i>) Weighted Methods per Class	[54]	Complexity	It is the sum of the <i>MCC</i> for all methods in a class
Requirements-level Metrics			
(<i>NC</i>) Number of Classes	[55]	Size	It is the number of classes implementing a requirement
(<i>CDC</i>) Requirements diffusion over components	[55]	Scattering	It is the number of classes that contribute to the implementation of the target requirements, among those of the application
(<i>CDC+</i>) <i>CDC</i> with similarity	-	Scattering	It is a variant of <i>CDC</i> in which the contribution of each class is weighted according to the similarity of each class with the requirements definition
(<i>ShR</i>) Shared among Requirements	[56]	Tangling	It expresses the degree of classes that implement a requirement and that are shared with, at least, another requirements of the application
(<i>ShR+</i>) <i>ShR</i> with similarity	-	Tangling	It is a variant of <i>ShR</i> in which the contribution of each class is weighted according to the similarity of each class with the definition of the requirements under analysis
(<i>IN</i>) Contained Requirements	[56]	Inheritance	It is the number of requirements whose implementation is entirely "contained" in target requirements

the maintainability index for each application class and requirement by aggregating metrics in Table 1 according to their weights as follows:

$$MI(element) = \frac{\sum_{\mu \in Metrics} (W_{\mu} * O_{\mu})}{\sum_{\mu \in Metrics} (W_{\mu})} \quad (1)$$

Metrics is the set of metrics in the models, W_{μ} is the weight of the metric μ and O_{μ} is 0 if element is not an outlier for the metric μ , while it is 1 if element is an outlier for μ . For instance, if a class $c1$ is an outliers only for CBO, LOCs, and DIT then $MI(c1) = 5/13 = 0.385$. Therefore, $c1$ has 0.385 (i.e., 38.5%) as index.

- Maintainability index for classes.* To compute the maintainability index $MI_{class}(c)$, we use only class-level metrics.
- Maintainability index for requirements.* To compute the maintainability index $MI_{req}(r)$ for each requirement r , two (sub-) indexes have to be computed, namely $MI_c(r)$ and $MI_R(r)$. The former is computed by averaging the index MI_{class} of all classes implementing r , while the latter is computed by applying requirements-level metrics to the classes implementing r . In both the cases, similarity links (SimilarityLinks) are needed. Then, by averaging $MI_c(r)$ and $MI_R(r)$ for each requirement r , we compute the overall index by considering, at the same time, the code implementing r in isolation and its implementation with respect to other requirements [52].

3.2.3 Prioritizing classes and requirements

Classes and requirements are prioritized according to their estimated index (i.e., MI_{class} and MI_{req} , respectively). According to this prioritization, we give a different weight to each code class and requirement of the application when

TABLE 2
Software Quality Model (the Metrics and weights W)

Maintainability: Software Quality Models									
Class-level Model									
Metric	CBO	RFC	LCOM	LOCs	NOM	DIT	NOC	MCC	WMC
W	2	2	2	1	1	2	1	1	1
Requirements-level Model									
Metric	NC	CDC	CDC+	ShR	ShR+	IN			
W	2	2	2	1	1	1			

computing their coverage for a test case. In detail, the weight we use to prioritize the coverage of application code (see below w_c in Equation 2) and the weight we use to prioritize the coverage of application requirements (see below w_r in Equation 4) are related to the ranking of classes and requirements obtained with MI_{class} and MI_{req} respectively.

3.3 Multi-Objective Reduction

The evaluation of all the possible test suite reductions (ReducedTestSuite) on several dimensions could be expensive also in case of non-large test suites. Hence, we adopted the formulation of test suite reduction as a multi-objective optimization problem, introduced, e.g., by Yoo et al. [16]. In a multi-objective optimization algorithm more than one objective functions (concerning different dimensions of the problem) have to be optimized simultaneously to achieve reasonably good solutions. In this kind of problem, there is not a unique solution that optimizes each considered dimension, but a set of potentially (sub-)optimal and equally good solutions exists. Such solutions compose *Pareto fronts* [16], [57]. A Pareto front includes solutions that are non-dominated, according to the considered dimensions, by any other solution within the space of all possible solutions. Given two solutions A and B, A dominates B if B is inferior to A in at least one of the considered dimensions.

To face multi-objective optimization problems, a number

of evolutionary algorithms have been proposed: NSGA-II [22], Strength Pareto Evolutionary Algorithm 2 (SPEA-2 [58]), particle swarm optimization (PSO [59]), and simulated annealing (SA [60]). Although different algorithms could be used, we resort to the use of NSGA-II. We opted for this algorithm because it optimizes conflicting objectives and because it has been widely and successfully used in software engineering and software testing (e.g., [16], [51], [61], [62]). NSGA-II is founded on the following concepts:

- genetic modification, for changing (evolving) solutions;
- elitism, for allowing the algorithm to converge towards better solutions;
- non-domination, for ranking solutions according to objective functions, thus selecting and evolving most promising solutions;
- crowding distance, for increasing the diversification of solutions.

NSGA-II applies a set of genetic operators (i.e., mutation, crossover, selection, and replacement) to iteratively evolve an initial population of reduced test suites. The evolution is guided by an objective function that evaluates each reduced test suite along the chosen objectives, three in our case. In each iteration, the population of the best alternative solutions is generated from an evolved population. In terms of complexity, the time complexity of the NSGA-II algorithm is $O(o \cdot p^2)$, where o is the number of objectives and p is the population size, while its space complexity is $O(p^2)$ [22]. The obtained Pareto front contains non-dominated solutions, in our case, it represents an optimal trade-off between structural, functional, and cost dimensions. To reduce test suites, the used implementation of the NSGA-II algorithm is based on the JMetal framework [63]. This implementation relies on the following steps:

- 1) A set (named initial population) of possible test suite reductions are randomly selected among all the possible test suite reductions of S .
- 2) The population is then evolved by applying a set of genetic operators, such operators change (i.e., evolve) a bit the reduced suites of the population at each iteration. The suites in the population are evaluated with respect to each of the three objectives (code and requirements coverage and execution time) and the fitness of each generated suite is computed considering the number of suites in the population dominated by that suite. Solutions are ranked according to each objective function. This allows the algorithm to promote the set of non-dominated suites in the population, thus being able to identify and preserve the “better” reduced suites of the population. Additionally to the fitness value, the crowding distance is calculated for each solution. The crowding distance for a solution is the average distance between the solution and its neighbor solutions. In other terms, it measures how close a solution is to all its neighbors. High crowding distance value increases

the population diversity.

- 3) After a number of iterations, the evolution is concluded and a set of optimal reduced suites are generated. That is, the population is expected to be stable in terms of size. The user/tester has to select one of these suites according to her testing needs and project resources. In particular, the tester can inspect the Pareto front to find the adequate trade-off by having: a test suite that balance code coverage, requirements coverage, and execution cost or a test suite that maximizes one/two dimension/s penalizing remaining one/s.

The *MORE+* set-up of the used NSGA-II instance can be summarized as follows:

- **Solution Encoding:** A solution is a possible reduced test suite (S_{red}) of the whole test suite S . The solution space for test suite reduction is given by all S_{red} . For instance, given a test suite S composed of the test cases $t1$, $t2$, and $t3$, possible reduced suites S_{red} are: $\langle t1 \rangle$, $\langle t2 \rangle$, $\langle t3 \rangle$, $\langle t1, t2 \rangle$, $\langle t1, t3 \rangle$, and $\langle t2, t3 \rangle$, respectively. A reduced test suite is encoded as a vector of bits, where each bit is 1 if the test case is part of the suite, 0 otherwise. The maximum number of test cases contained in the reduced suite could be forced by testers, i.e., it is a parameter the tester can customize (e.g., 30% of size of S), alternative it is determined by the algorithm.
- **Initialization:** We randomly initialize the starting population by selecting test suite reductions among all the possible ones.
- **Genetic Operators:** We used the bit-flip mutation operator, where one randomly chosen element of the solution is changed. The used crossover operator is the conventional one-point crossover, in which the crossover point is selected randomly in the two solution parents. The crossover point is the point from which the two tails of parent solutions are then recombined. Mutation and crossover operators are adopted to find new candidate solutions having similarities with selected solutions.

The adopted selection operator is the conventional binary tournament, in which a solution is selected if its ranking is better than the one of other solutions or, in case the ranking is the same, if its crowding distance is greater than the ones of other solutions. The selection operator is adopted to possibly improve the current population by exploiting the better solutions it contains.

As replacement operator, we applied the replace worst (elitism) strategy to incorporate new candidate solutions into the current population.

- **Objective Functions:** The aim is to maximize the three considered objectives. Each candidate solution in the population (each reduced test suite) is evaluated by our objective functions: the overall *source code coverage* of a reduced suite S (namely $cumCCov(reds)$), the overall *application requirements coverage* of a reduced

suite S (namely $cumRCov(reds)$), and the overall *tests case execution cost* of a reduced suite S (namely $cumCost(reds)$); larger values of the functions are desired. The fitness of a solution is evaluated based on the number of other solutions it dominates as described before.

In the next subsections, we present in detail metrics just mentioned.

3.3.1 Computing Metrics for test case evaluation

- **Code.** Fault detection capability cannot be known before executing test cases. Therefore, we have to resort to the “potential” fault detection capability of a test suite. Traditionally, it can be estimated by considering the amount of source code covered (CodeCovered) by the test cases of a test suite at run-time [64], [65], [66]. A test case that covers a larger set of code statements should have a higher fault detection capability (i.e., potentially more faults are revealed) than a test case that covers a smaller set of statements. Hence, code coverage is largely used in the literature as a proxy to estimate fault detection capability. However, a recent study [15] (conducted on 5 Java applications) showed that a strong correlation exists between code coverage and test suite effectiveness i.e., given two suites for a system, the one that better covers the system code is expected to have higher fault revealing capability. The study, however, showed also that this is not completely true if the suite size is controlled/limited, e.g., in case of test suites reduction. This result contributed to motivate us in considering code coverage and additional dimensions, such as cost and functional coverage, when evaluating application coverage.

In this work, we assume to have implementations of JUnit test cases (TestCases) and define the *Weighted Code Coverage* measure $WCCov(t)$, for a given test case, as follows:

$$WCCov(t) = \sum_{s \in Statements} \begin{cases} w_c & s \in CodeCovered \\ 0 & otherwise \end{cases} \quad (2)$$

where the set *Statements* contains code statements (Code); *CodeCovered* is the set of statements covered by the test case t ; w_c ($0 \leq w_c \leq 1$) is a weight associated to code statements s or blocks of them (e.g., all statements of a class). The weight w_c is defined according to testing needs. In our previous work [21], this weight was 1 for all parts of the application code, namely we equally weighted each code statement. Conversely, we propose in this paper a metric-based approach (see Section 3.2) to automatically identify such a weight (Weights) by estimating the maintainability of each class. This allowed us to prioritize classes according to their maintainability when computing coverage. In other terms, w_c is related to the ranking defined by means of MI_{class} (see Section 3.2).

We define the overall code coverage of a suite S , namely $cumCCov(S)$, as the sum of the code coverage of all the test cases of the suite S .

$$cumCCov(S) = \sum_{t \in S} WCCov(t) \quad (3)$$

- **Requirements.** The capability of a test case to exercise application requirements depends on: (i) the amount of requirements covered by a test case; (ii) the relevance of covered requirements; and (iii) the existing dependency/relationship among requirements. To deal with these aspects, we defined the *Weighted Requirements Coverage with Dependencies* measure $WRCovD(t)$ as:

$$WRCovD(t) = \sum_{r \in ReqsCovered} w_r * \left(\sum_{r_\beta = r \in Reqs} w_{rD}(r, r_\beta) \right) \quad (4)$$

Reqs is the set of requirements of the application under test (Reqs), *ReqsCovered* (i.e. $\subseteq Reqs$) is the set of requirements covered by t , while r and r_β are application requirements, and w_r is a weight associated to each requirement that assumes values in between 0 and 1. In our previous work [21], this weight was 1 for each application requirement, namely we equally weighted each requirement. Conversely, here, we propose a metric-based approach to automatically identify such a weight (Weights), thus being able to order requirements according to their estimated fault-proneness. In other terms, w_r is related to the ranking defined by means of MI_{req} (see Section 3.2). Finally, w_{rD} measures the strength of each requirement relationship (rD) and, in particular, between r and other requirements of the application r_β . Given the pairs of requirements r_α and r_β , this strength is computed as follows:

$$w_{rD}(r_\alpha, r_\beta) = \frac{w_{req}(r_\alpha, r_\beta) + w_{code}(r_\alpha, r_\beta)}{2} \quad (5)$$

$w_{rD}(r_\alpha, r_\beta)$ tends to 1 if a strong relationship exists between r_α and r_β , i.e., both textual description and implementation strongly overlap. $w_{rD}(r_\alpha, r_\beta)$ tends to 0 if no relationship exists between r_α and r_β . The functions $w_{req}(r_\alpha, r_\beta)$ and $w_{code}(r_\alpha, r_\beta)$ computes weights of the relationships of requirements r_α and r_β and their code implementations:

$$w_{req}(r_\alpha, r_\beta) = IRSimilarity(r_\alpha, r_\beta) \quad (6)$$

$$w_{code}(r_\alpha, r_\beta) = \frac{overlapClasses(r_\alpha, r_\beta)}{totalClasses(r_\alpha, r_\beta)} \quad (7)$$

where $w_{req}(r_\alpha, r_\beta)$, inferred by LSI, provides an indication about the possible link between requirements r_α and r_β , while $w_{code}(r_\alpha, r_\beta)$ computes the ratio between the portion of code that is in common between the implementations of r_α and r_β (*overlapClasses*) and the whole code implementing them (*totalClasses*). Similarity links (SimilarityLinks) are used to identify the portion of source code (Code) that is in common between the implementations of r_α and r_β . $WRCovD(t)$ is expected to give more relevance to the test cases covering requirements having strong relationships with a high number of other requirements that is to the test cases exercising “key” requirements.

We define the overall requirement coverage of a suite, $cumRCov(S)$, as the sum of the requirements coverage of all test cases of the suite.

$$cumRCov(S) = \sum_{t \in S} WRCovD(t) \quad (8)$$

- **Execution cost.** It is approximated by the time required to execute a given test case t (available in `:ExecutionTime`). We defined: $Cost(t)$ as the time to execute t , and $Cost(S)$ as the overall cost of S , computed as the sum of the executions of all its test cases. In our case, for the way in which we defined our optimization problem (i.e., as a maximization problem,), we define $RelativeCost(t)$, of a test t in the suite S , as follows:

$$RelativeCost(t) = Cost(S) - Cost(t) \quad (9)$$

Hence, the overall cost of a suite S , namely $cumCost(S)$, is the sum of the relative cost of its test cases.

$$cumCost(S) = \sum_{t \in S} RelativeCost(t) \quad (10)$$

4 DESIGN OF THE EMPIRICAL EVALUATION

We investigated the following two main research questions:

RQ1: Is *MORE+* effective in reducing test suites?

RQ2: Is *MORE+* efficient in reducing test suites?

To answer these research questions, we considered seven baseline approaches to compare our proposal with. We selected these baselines because they represent the standard for comparison in the test suite reduction field (e.g., [2]). These baselines are:

- The original test suite S (named Full).
- Four traditional test suite reduction approaches named: Harrold Gupta Soffa (HGS), delayed greedy (DGR), traditional greedy (GRD), and 2-optimal greedy (2OPT) [8], [9], [10], [26]. In our study, we adopted the two-objective extension of these approaches that considers both code coverage and execution time. In detail, the adopted version of HGS, DGR, GRD and 2OPT uses the following ratio metric to evaluate each test case when reducing test suites:

$$ratio(t) = \frac{codeCoverage(t)}{executionCost(t)} \quad (11)$$

where t is the test case under evaluation, $codeCoverage(t)$ is the code covered by t when and $executionCost(t)$ is the time required to execute t . This metric allows the approaches to reduce test suites by preserving test cases that cover the most requirements per unit of cost. Smith et al. [8], [26] observed, in fact, that such a variant outperforms the traditional single-objective version.

- NSGA-II algorithm (named *NSGAIIda*). It is a standard NSGA-II approach that considers: code coverage and execution cost. The implementation of the NSGA-II algorithm we used is based on the JMetal framework [63].
- An extended version of a traditional greedy approach (named GR3D). It is a three-dimension version of the traditional greedy approach (i.e., GRD) that considers: execution cost and code and requirement coverage.

In our empirical study, we considered the following quantitative criteria (or constructs) to assess *MORE+* results and

TABLE 3
Objects under study

App.	Size (LOCs)	Test Cases	Reqs	Faults
LaTazza	2k	33	10	12
AveCalc	2k	47	10	15
CommonsProxy	5k	179	10	10
DBUtils	5k	225	12	14
iTrust	15k	919	15	21
CommonsCodec	17k	608	19	20
JTidy	20k	289	25	15
Woden	22k	263	24	19
Log4J	25k	1029	24	20
JXPath	25k	386	20	20
CommonsIO	25k	859	18	20
CommonsBcel	30k	75	20	20
CommonsBeanUtils	32k	1556	26	22
xmlGraphics	34k	196	24	15
xmlSecurity	40k	92	23	15
CommonsCollections	50k	798	17	20
Pmd	55k	698	20	20
CommonsLang	60k	2307	16	20
Jabref	70k	213	31	20
Xerces	138k	376	20	20

to compare these results with those obtained by applying the baseline approaches.

- **Reduction in test-suite size** (RQ1) concerns the size of reduction degree between the number of test cases of S (original suite) and S_{red} (reduced suite).
- **Reduction in fault-detection capability** (RQ1) concerns the capability of S_{red} in revealing faults with respect to S .
- **Reduction in test-suite execution time** (RQ1) concerns the reduction factor in terms of execution time between S and S_{red} .
- **Diversity** (RQ1) concerns the difference among reduced test suites (with respect to their test cases).
- **Artifact coverage** (RQ1) indicates the capability of S_{red} to cover applications artifacts. It gives an idea about how well a test suite covers both code and requirements.
- **Reduction Approach Cost** (RQ2) concerns the cost to reduce test suite.
- **Cost-Effectiveness** (RQ2) concerns the cost-benefits of a test suite reduction approach.

To complete our data analysis, we additionally analyzed: the impact of the suite size in the achieved results (“Are large reduced suites more effective than small ones?”), the composition of Pareto fronts produced by *MORE+* (“What is the test effectiveness of different suites in Pareto fronts built by *MORE+*?”), and the possible effect of co-factors on obtained results (“Is there any co-factor impacting on the achieved results?”). In this last analysis, we considered: (i) application artifacts, (ii) distribution of faults; and (iii) capability of test cases in revealing faults.

4.1 Experimental Objects and Measures

We used 20 Java applications from different domains as experimental objects. In Table 3, we report descriptive statistics of these applications: LOCs (excluding comments and blank lines) and number of test cases, requirements, and (real) faults. These Java applications were chosen because

representative of open-source software and because the availability of the artifacts needed to execute *MORE+*. The size of these applications ranges in between small and large.

We considered the following measures (or dependent/response variables) to estimate the constructs of interest in our empirical study. For readability reason, we report the name of each construct and the measure/s used to estimate such a construct.

- **Reduction in test-suite size.** Given a test suite S and a reduced suite S_{red} , we compute RS as follows:

$$RS(S_{red}) = \frac{|S| - |S_{red}|}{|S|} * 100 \quad (12)$$

$RS(S_{red})$ represents the percentage of test cases of S that are not in S_{red} . RS values range in between 0 and 100. A value close to 0 indicates that in S_{red} there are almost the same test cases as those in S . A value close to 100 indicates that the greater part of test cases of S have been removed.

- **Reduction in fault-detection capability.** For this construct, we used two measures: RF and F . RF measures the reduction in the capability of detecting faults. It is computed as the ratio of fault-detection capability of S_{red} on fault-detection capability of S . In particular, given a reduced suite S_{red} of S , RF is computed as follows:

$$RF(S_{red}) = \frac{|F| - |F_{red}|}{|F|} * 100 \quad (13)$$

where F denotes the set of faults revealed by a suite S and F_{red} the set of faults detected by the reduced test suite S_{red} . Therefore, $RF(S_{red})$ indicates the percentage of faults not detected. RF ranges in between 0 and 100. If RF assumes 0 as the value, it means that all faults detected by S are also detected by S_{red} . That is, there is no reduction capability of S_{red} in revealing faults. On the other hand, if RF assumes 100 means as the value it means that no faults were detected by S_{red} . The lower the RF value, the better it is.

We also consider F , the suite effectiveness in detecting faults. F is the absolute number of faults a given test suite is able to reveal. The higher the value of F , the greater the effectiveness of the test suite is.

- **Reduction in test-suite execution time.** For this construct, we used the measure RT . It measures the reduction factor for the execution time of test cases in S_{red} with respect to S . RS and RT could be related one another. However, it could happen that two reduced suites of S having the same number of test cases (i.e., size) require different execution times (e.g., test cases in a reduced test suite need more time to be executed). RT of a reduced suite S_{red} is computed as follows:

$$RT(S_{red}) = \frac{time(S) - time(S_{red})}{time(S)} * 100 \quad (14)$$

where $time(S)$ is the time needed to execute S and $time(S_{red})$ is the execution time of S_{red} . The values of RT range in between 0 and 100. The higher the value, the greater the reduction of time to execute regression testing

with respect to S is. Therefore, the greater the value the better it is.

- **Diversity.** To estimate this construct we used Div . It measures the diversity of test cases composing a pair of (reduced) test suites S_1 and S_2 . It is computed as the ratio of the number of test cases shared between two reduced suites, with respect to the number of test cases in Full. High value of $Div(S_1, S_2)$ means that suites have a high number of test cases in common. The higher the $Div(S_1, S_2)$ value, the lower the diversity between S_1 and S_2 is.

- **Artifact coverage.** We exploit two measures to assess the construct Artifact coverage of a suite S : $Code_Cov(S)$ and $Reqs_Cov(S)$. The former measures the number of executed code statements exercised at least once by test cases of S , while the latter the number of requirements exercised at least once. The higher the $Code_Cov$ value, the higher the code coverage is. Similarly, the higher the $Reqs_Cov$, the higher the requirements coverage is. For both measures, the higher the values, the better it is.

- **Cost of the reduction approach.** Test suite reduction approaches might be effective in reducing test suites but they can be expensive to use, thus not reducing regression testing cost. For instance, in the case of the proposed reduction approach a concern can be related to the cost of the recovery of similarity links among software artifacts: how much it influences both viability and efficiency of our approach? We hence evaluated and measured $Cost_Red$, the absolute cost of applying reduction approaches in terms of time required to apply each approach.

- **Cost-Effectiveness.** The absolute cost of each reduction approach needs to be considered with respect to the actual use of the reduction approach in realistic scenarios, thus being able to evaluate the cost-effectiveness degree of the reduction approach ($CostEff$). To this respect, cost models have been proposed in the literature to evaluate the cost of a test suite reduction approach (e.g., [67], [68]). In our study, we exploited the cost model proposed by Malishevsky et al. [68]. In this model, the cost-effectiveness of a reduced suite is characterized by two main aspects: on the one hand the cost of executing the reduced suite and on the other hand the cost of omitting the detection of some faults that are not identified by the reduced suite. Low values for a given reduced suite with respect to other suites represent a better cost-effectiveness for that suite. In particular, given S and S_{red} (used to test g versions of an application), we can define: (i) $Ca(S)$ as the cost of the analysis conducted to prepare test suite execution and reduction (e.g., textual-based similarity recovery links); (ii) $Ce(S)$ as the execution cost of test cases; (iii) $Cc(S)$ as the cost of result checking; (iv) $Cr(S)$ as the cost for the execution of a test suite reduction approach (i.e., effort/time required to obtain the reduced suite); (v) $Cm(S)$ as the maintenance cost of a test suite (e.g., effort and time for fixing a suite); and (vi) $Cf(F_k(S) \setminus F_k(S_{red}))$ as the cost of omitting faults by not selecting the set $S \setminus S_{red}$, where F_k is the set of faults detected in the release k . Hence, the cost C to retest-all strategy is the cost to execute Full, check results, and perform maintenance operations. This cost is

computed as follows:

$$C = g * Ce(S) + g * Cc(S) + g * Cm(S) \quad (15)$$

Instead, the cost of a given reduction approach (i.e., where only a subset of S is executed and preserved to test next application releases) is:

$$CostEff = Ca(S) + g * Ce(S_{red}) + g * Cc(S_{red}) + Cr(S) + \sum_{1 \leq k \leq g} (Cf(F_k(S) \setminus F_k(S_{red}))) + g * Cm(S_{red}) \quad (16)$$

We excluded from our analysis the cost of: (i) results checking $Cc(S)$ and (ii) suite maintenance $Cm(S_{red})$. This was needed because this information was not available. Furthermore, we computed the cost of omitting faults $Cf(F_k(S) \setminus F_k(S_{red})) = fc * |Fnd_k|$, where Fnd_k is the set of faults a reduced suite does not detect and fc is the cost of omitting such faults. We measured such a cost function as a constant time (120 seconds) required to detect and fix bugs in a post-release version of an application. This is customary in the literature [67], [68].

4.2 Procedure

For each experimental object, we applied the following procedure:

- 1) *Collecting artifacts*. We collected the following artifacts: application requirements, source code, and test cases.
- 2) *Recovering textual similarity*. To recover similarity links among software artifacts, we used the following set-up for LSI: $k=300$; *constant threshold*=0.1.
- 3) *Running MORE+ and baseline approaches*. We ran *MORE+* and *NSGAI_{2d}* with the following set-up: *population size*=2*“test suite size”; *crossover probability*=0.9; *mutation probability*=1/“test suite size”; *number of iterations*=1000. Since *MORE+* and *NSGAI_{2d}* have a non-deterministic behavior, we ran them 10 times and collected all the generated solutions.
- 4) *Reproducing actual faults*. In the last column of Table 3, we report the number of faults we injected/seeded in each experimental object. Fault injection was accomplished by a researcher involved neither in the approach definition nor in the study execution. Each seeded fault reproduced an actual fault described by actual users and developers in the application issue tracking system and it has been reproduced into the original source code of the application. Therefore, we had a faulty version of an application for each seeded fault. The number of versions for each application is that shown in the column *Faults* of Table 3.

For each application, we randomly selected the faults to be seeded taking into account among the ones inserted in the issue tracking system. In detail, we analyzed failures described in a bug-tracker and used information describing a fault associated to that

(a)

```

1 | Index: src/main/java/org/apache/commons/dbutils/BasicRowProcessor.java
2 | -----
3 | --- src/main/java/org/apache/commons/dbutils/BasicRowProcessor.java (Revision 1597126)
4 | +++ src/main/java/org/apache/commons/dbutils/BasicRowProcessor.java (ArbelazkoPie)
5 | @@ -20,6 +20,7 @@
6 | import java.sql.ResultSetMetaData;
7 | import java.sql.SQLException;
8 | import java.util.HashMap;
9 | +import java.util.LinkedHashMap;
10 | import java.util.List;
11 | import java.util.Locale;
12 | import java.util.Map;
13 | @@ -181,7 +182,7 @@
14 |     * key.toString().toLowerCase()
15 |     * </pre>
16 |     */
17 | - private static class CaseInsensitiveHashMap extends HashMap<String, Object> {
18 | + private static class CaseInsensitiveHashMap extends LinkedHashMap<String, Object>
19 |     /**
20 |     * The internal mapping from lowercase keys to the real keys.
21 |     */

```

(b)

Fig. 3. DbUtils: Description of the bug DbUtils-114 (a) and the patch to fix it (b)

failure to analyze the code of the application and to get information about how to restore that fault. For instance, Figure 3(a) shows an example of a fault description posted in the DbUtils bug-tracker² (bug id=DbUtils-114) and raising a bug (in DbUtils version 1.5) concerning the order of the information result of a query. While Figure 3(b) shows the patch (code fragment) used to fix it (in DbUtils version 1.6). By the analysis of this information we observed that the fault compromises the behavior of a relevant application requirement (classified as “Major bug”) and we got information about the place where this fault occurred in the application code. We could then reproduce the fault in the application. The approach used to inject faults is very costly in terms of man-hours. However, it has the merit to reproduce actual faults in source code. This is why this approach is well known and widely adopted in the literature (e.g., [69], [70]).

- 5) *Executing test suites and collecting data*. We executed test suites (reduced and whole) on the faulty versions of the experimental objects and collected values for estimating the considered quantitative criteria. It is worth noting that, in this phase, we consider at least 20 reduced test suites for each approach.
- 6) *Analyzing data*. We analyzed collected data. We performed statistical analyses when needed. In the case of the evolutionary approaches (i.e., *MORE+* and

2. <http://issues.apache.org/jira/browse/DBUTILS>

NSGAII_{2a}), sets of solutions in each run were produced (solutions composing Pareto fronts). We considered all these solutions in our data analysis.

4.3 Null Hypotheses and Statistical Tests

We tested the following hypothesis:

NH_m - **there is no difference** in values of m (e.g., RS, RF, RT and F) computed on test suites reduced by applying *MORE+* and baselines.

This hypothesis is described as a single parametrized null hypothesis and it is two-sided because we could not do any postulation on which approach is the best on each considered dependent variable. Our parametrized null hypothesis corresponds to a number of null hypotheses, one for each variable (e.g., NH_{RS}). In case the statistical test will reject a null hypothesis, we can accept the alternative one (e.g., *there is a statistically significant difference* in values of RS computed on test suites reduced by applying *MORE+* and the baseline under analysis).

To test the null hypotheses, we applied the Kruskal-Wallis test [71]. This statistical test is a non-parametric test that allows checking whether a set of observed independent samples originate from the same distribution, i.e., the null hypothesis is that medians of samples are equal, while the alternative hypothesis is that at least one sample median among the considered ones is different. The Kruskal-Wallis Test is analogous to the parametric one-way Analysis of Variance (ANOVA) test, but it does not assume a normal distribution of the sample residuals (being it a non-parametric test). If the Kruskal-Wallis test reveals a statistically significant difference (i.e., the null hypothesis is rejected), we performed a post-hoc analysis. In particular, we perform a pairwise comparisons among the results achieved for test suites reduced by applying *MORE+* and each baseline approach (e.g., there is no significant difference in values of a variable m between *MORE+* and one of the baselines). To this aim, we used a non-parametric two-sided Mann-Whitney test. The Mann-Whitney test is a non-parametric test that allows comparing two samples of ordinal data to check the null hypothesis that samples come from the same population against an alternative hypothesis, that samples come from different distributions. We opted for a two-sided Mann-Whitney test because we could not support any specific direction of alternative hypotheses. By means of the Mann-Whitney test we evaluated the statistical significance of the difference between two approaches, i.e., if the observed difference is actual or due by chance. To have an indication of the relevance or such observed difference, however, we also measured the effect size by applying the Vargha and Delaney test (\hat{A}_{12}), as presented by Arcuri [72]. \hat{A}_{12} ranges between 0 and 1: 0.5 indicates that two approaches under test are stochastic-equivalent, while values closer to 0 or 1 indicate that a stochastic difference exists. In our case, we used \hat{A}_{12} to compare the probability of achieving better results for two suite reduction approaches A (i.e., *MORE+*) and B (i.e., a baseline approach). For instance, for each

pair of reduction approaches A and B, we applied \hat{A}_{12} for F and RF. For F, that we aim to maximize, \hat{A}_{12} equal to 0.5 indicates that the two approaches are equivalent; and $\hat{A}_{12} > 0.5$ indicates that A has higher probability of getting better solutions than B. Instead, for RF that we should minimize, $\hat{A}_{12} < 0.5$ indicates that A has higher chances of getting better solutions than B. To further analyze the relationship between the two measures, in specific cases, we adopted the Chi-squared test of independence. The Chi-squared test of independence investigates the existing independence between two variables x and y : “they are independent if the probability distribution of one variable is not affected by the presence of another”.

To evaluate the impact of possible co-factors, we applied a two-way permutation test [73]. Our null hypothesis is:

$NH_{co-factors}$ - **there is not a significant impact** of the considered co-factor/s on measured values for each dependent variable.

The two-way permutation test is a non-parametric equivalent of the two-way ANOVA. This test allows us to identify the presence of any dependency or interaction among the considered variables. The permutation test does not make any assumptions about the sample distribution and gives a way to compute the sampling distribution of statistic tests. The underlying idea is that if the null hypothesis is true, changing the exposure (e.g., by randomly shuffling it) will have no effect on results. Permutation tests are especially used when data are sampled from unknown distributions, when the size of samples is small and when outliers could be present. In case of number of the variables are small and their variability is low, permutation tests can exhaustively consider all permutations. However, more frequently, number and variability of variables are not trivial, thus permutation tests could be applied considering a subset of all possible permutations by adopting some stopping criteria. The number of considered permutations is relevant to have enough confidence on the achieved results. It is worth mentioning that an increase of the number of permutation increases the effort required to get the output of the test, thus “confidence” and “effort” have to be balanced. Two approaches can be adopted: (i) stopping when a maximum number of permutation has been considered (often this number is 1000 or 5000) and (ii) stopping when the estimated standard deviation of the p-value is under a given amount, often 10%, of the estimated p-value, namely the Anscombe’s criterion [74]. We stopped the permutations according to the Anscombe’s criterion or, in any case, when 5000 permutations have been reached. Additionally, for the co-factors analysis, we also performed a Spearman correlation test [75]. It is a non-parametric test and measure rank correlation to check if the relationship between two variables can be described by using a monotonic function.

In all performed statistical tests, we decided to accept a probability of 5% of committing a Type-1-Error [71], the incorrect rejection of a true null hypothesis (a “false positive”). A null hypothesis is hence rejected only if the p-value of a statistical test is less than 0.05. However,

when repeating tests, we are increasing the risk of finding significant differences between samples by chance. Therefore, in such a case, a compensation method of repeated statistical tests has to be applied. We used the Benjamini-Hochberg correction [76]. This method lets us control the false discovery rate (proportion of false positives among the set of rejected hypotheses) and it is well known and widely used in empirical studies (e.g., [77]). Among the correction methods (e.g., Bonferroni and Holm), the Benjamini-Hochberg is the least stringent method and it is known to provide a balance between discovery of statistically significant results and limitation of false positive [76].

4.4 Threats to validity

Despite our efforts to mitigate as many threats to validity as possible, some are unavoidable. For example, a possible threat to the validity of our results is the set of faults and their distribution in the source code of the experimental objects. Different sets of faults could lead to different results. To deal with this kind of threat, we reproduced actual faults. The used experimental objects represent another possible threat to the validity of the results: all of them are applications developed in open-source projects.

The set-up of the study represents another possible threat to the validity of the results. For instance, the number of runs as well as the tuning parameter values chosen for both the recovery of links among software artifacts and the multi-objective algorithms could potentially affect results (in positive or negative fashion).

Threats to the validity of the results could be due to the performed statistical tests. We used statistical tests that were well known for their robustness and sensitiveness. When computing repeated tests, we apply compensation.

5 RESULTS

In the following subsections, we present the obtained results according to the considered constructs.

5.1 Reduction in test-suite size

In Table 4, we report descriptive statistics for RS and the results of the statistical analyses. The p-values obtained by applying the Kruskal-Wallis and the Mann-Whitney tests are also reported, as well as the \hat{A}_{12} effect size. The Kruskal-Wallis test allowed rejecting NH_{RS} (p-value < 0.001). *MORE+* and *NSGAII_{2d}* obtain comparable RS values, i.e., no statistical difference was present. *MORE+* identified test suites larger than those identified by HGS, DGR, GRD, 2OPT and GR3D. Descriptive statistics show that the median value for *MORE+* of RS is 64% (see Table 4) and the (average) RS medians for other approaches (HGS, DGR, GRD, 2OPT, GR3D) is 79.7%. This result is also confirmed by the \hat{A}_{12} effect size (about 0.3 in favor of traditional reduction approaches). Among the traditional approaches, GR3D tends to reduce slightly less than others. Results confirm that, since test suite reduction approaches based on evolutionary algorithms (*MORE+* and

TABLE 4
Descriptive statistics and statistical analysis results (* indicates a statistical significant difference by applying the Benjamini-Hochberg correction when using the Mann-Whitney test)

	Min	Med.	Mean	Max	sd	Mann-Whitney <i>p</i> - value	\hat{A}_{12}
RS (Reduction in test-suite size)							
<i>MORE+</i>	32.7	64	65	90.6	15.9	-	-
2OPT	43.4	79.2	75.0	97.9	17.2	0.008*	0.32
DGR	49.3	80.4	78.2	97.9	14.7	0.003*	0.3
GRD	46.9	79.8	76.9	97.9	15.8	0.007*	0.32
HGS	48.8	80.4	78.1	97.5	14.7	0.003*	0.3
<i>NSGAII_{2d}</i>	50	65.3	67.8	84.8	10.1	0.712	0.49
GR3D	43.6	78.7	75.9	88.6	16.7	0.012*	0.33
Kruskal-Wallis p-value < 0.001							
F (Faults)							
<i>MORE+</i>	1	9	9.7	18	3.4	-	-
2OPT	1	8.5	7.9	19	4.3	0.04*	0.63
DGR	1	7	7.9	18	4.2	0.032*	0.64
GRD	1	7.5	7.5	19	4.4	0.011*	0.66
HGS	1	6.5	7.5	18	4.1	0.007*	0.67
<i>NSGAII_{2d}</i>	2	8	8.8	19	3.1	<0.001*	0.57
GR3D	1	7.5	7.4	19	4.6	0.014*	0.66
Kruskal-Wallis p-value < 0.001							
RF (Reduction in fault-detection capability)							
<i>MORE+</i>	0	46.6	45.1	93.3	18.9	-	-
2OPT	5	52.5	55.2	95	22	0.056	0.37
DGR	10	60	56.1	91.6	21.7	0.021*	0.34
GRD	5	57.5	58.1	95	22.8	0.013*	0.33
HGS	10	60	58.37	91.6	20.4	0.004*	0.31
<i>NSGAII_{2d}</i>	0	53.3	51	90	17.6	<0.001*	0.41
GR3D	5	57.5	58.4	95	25.2	0.019*	0.34
Kruskal-Wallis p-value < 0.001							
RT (Reduction in test-suite execution time)							
<i>MORE+</i>	13.9	61.4	59.2	94.8	19.7	-	-
2OPT	26.5	73.2	69.1	98.9	23.5	0.036*	0.36
DGR	26.9	74.3	69.4	98.4	23.1	0.037*	0.36
GRD	26.6	73.4	69.4	98.9	23.6	0.034*	0.36
HGS	27	74.8	70	98.6	23	0.028*	0.35
<i>NSGAII_{2d}</i>	30.1	61.4	62.5	82.7	10.5	0.338	0.47
GR3D	25.7	73.3	68.5	98.7	24.1	0.046	0.36
Kruskal-Wallis p-value = 0.0036							

NSGAII_{2d}) are explicitly balancing among different dimensions when reducing a test suite, they have a less suite reduction capability with respect to traditional approaches focused on two (HGS, DGR, GRD, and 2OPT) and three (GR3D) dimensions. Table 5 (top-left) details the (mean) values of RS obtained for each experimental object and test suite reduction approach. We can see that only on 8 out of 20 applications, approaches based on evolutionary algorithms tend to achieve better RS values as compared with traditional approaches. *MORE+* achieved results better than other (non-evolutionary) approaches on 5 applications, while it obtained results similar to those of *NSGAII_{2d}*.

5.2 Reduction in fault-detection capability

Results for the faults F (see Table 4) suggest that we can reject the null hypothesis NH_F (the Kruskal-Wallis test returned 0.001 as the p-value), while the results of both the Mann-Whitney and the \hat{A}_{12} (>0.57) tests suggest that *MORE+* achieved better results than other approaches (*NSGAII_{2d}* included). Table 5 (down-right) details the (mean) values of F obtained for each application and suite reduction approach. Table 4 reports also results obtained for RF. We can observe that results of the Kruskal-Wallis and the \hat{A}_{12} tests suggest that RF values had also independent distributions (p-value < 0.001) and that a significant effect (>0.3) in favor of *MORE+* exists. Test suites reduced by

TABLE 5
RS, RF, RT and F detailed values per application

APP	2OPT	DGR	GRD	GR3D	HGS	MORE+ (mean)	NSGAI12d (mean)	2OPT	DGR	GRD	GR3D	HGS	MORE+ (mean)	NSGAI12d (mean)
	RS							RF						
AveCalc	87.2	87.2	87.2	87.2	87.2	78.7	78.7	33.3	26.7	33.3	33.3	33.3	13.8	22
CommonsBcel	69.3	69.3	69.3	68	69.3	61.3	64.2	65	65	65	65	65	56.9	65.8
CommonsBeanUtils	80.8	82.1	81.3	80.2	82	73.8	73.8	31.8	31.8	31.8	36.4	31.8	33.5	36.6
CommonsCodec	92.1	93.3	93.3	92.9	93.3	64.5	64.5	90	80	90	95	85	48	50
CommonsCollections4	49.1	52	50	45.9	52	60	60	45	40	45	40	45	52.6	43.3
CommonsIO	49.2	59.5	55.4	53.9	59.4	60.1	50.1	50	55	55	55	55	48.3	44
CommonsLang	94.2	96.2	96.2	96.1	96.2	83.3	70	95	75	95	95	80	61.1	51.5
CommonsProxy	78.8	79.3	78.8	76.5	79.3	79.9	83.2	40	30	40	20	40	44.4	26.6
DbUtils	77.8	78.7	77.8	77.3	78.7	43.1	64.4	42.9	50	42.9	50	57.1	10.1	31.6
iTrust	79.8	82.8	81.7	80.5	82.7	71.8	52.9	71.4	66.7	71.4	71.4	71.4	59.7	40.8
Jabref	45.5	49.3	46.9	43.7	48.8	41.3	70.9	50	35	50	45	50	35.1	71.1
Jtidy	97.9	97.9	97.9	96.9	97.6	90.3	50.2	80	80	80	80	80	70.9	48.1
JXPath	80.3	81.6	80.8	81.1	81.6	82.9	82.9	55	60	60	60	60	59.7	57
LaTazza	93.9	97	97	97	97	73.1	78.8	66.7	91.7	91.7	91.7	91.7	18.5	25
Log4j	72	72.9	72.5	71.2	72.9	90.9	77.1	5	10	5	5	10	50	15.7
Pmd	83.5	86.2	85.8	85.5	86.2	69.7	69.8	75	60	75	80	60	68.5	78.3
Woden	58.9	59.7	59.3	57.4	59.7	32.7	61.7	47.4	47.4	47.4	47.4	42.1	22.4	58.3
Xerces	92	93.9	93.9	93.9	94.4	84.8	84.8	55	65	65	65	70	57.6	58.7
xmlGraphics	74.5	74.5	74.5	73.5	74.5	74.5	57.1	73.3	73.3	73.3	86.7	73.3	57.8	51.1
xmlSecurity	43.5	70.7	59.8	59.8	70.7	60.9	60.9	33.3	80	46.7	46.7	66.7	48.4	57
	RT							F						
AveCalc	79.8	79.3	79.8	79.3	79.8	64.4	64.1	10	11	10	10	10	12.9	11.6
CommonsBcel	70.4	70.4	70.4	69.3	70.4	61.9	65.9	7	7	7	7	7	8.6	6.8
CommonsBeanUtils	73.3	73.4	73.3	73.2	73.4	62.3	55.9	15	15	15	14	15	14.61	14
CommonsCodec	93.2	91.8	93.4	91.8	93.3	60.8	60.4	2	4	2	1	3	10.4	10
CommonsCollections4	45.4	45.4	45.4	42.6	45.9	57.8	56.5	11	12	11	12	11	9.4	11.3
CommonsIO	47.1	48.9	47.3	48.4	50.6	60.2	48.8	10	9	9	9	9	10.33	11.2
CommonsLang	98.9	98.4	98.9	98.7	98.6	83.2	73.5	1	5	1	1	4	7.76	9.7
CommonsProxy	34.9	35.3	34.9	28.2	35.7	39.9	49.1	6	7	6	8	6	5.56	7.3
DbUtils	87.6	87.5	87.6	87.4	87.7	19.1	59.4	8	7	8	7	6	12.5	9.57
iTrust	93.2	93.3	93.2	91.9	93.4	69.9	50.8	6	7	6	6	6	8.44	12.4
Jabref	47.3	46.5	47.6	46.9	50.7	42.4	70.8	10	13	10	11	10	13	5.7
Jtidy	97.3	97.3	97.3	96.3	96.9	90.2	50.7	3	3	3	3	3	4.35	7.7
JXPath	45.91	47.70	46.2	46.7	47.7	72.1	74.3	9	8	8	8	8	8.5	8.6
LaTazza	92.4	96.1	96.2	96.1	96.1	65.1	72.9	4	1	1	1	1	9.78	9
Log4j	26.5	26.9	26.6	25.7	27.1	84.3	41.1	19	18	19	19	18	10	16.8
Pmd	88.3	86.4	88.3	87.9	88.1	71.6	70.8	5	8	5	4	8	6.2	5.2
Woden	58.5	57.4	58.5	57.7	58.7	30.5	61.4	10	10	10	10	11	14.7	7.9
Xerces	73.1	75.1	73.5	73.5	76.2	67.21	69.3	9	7	7	7	6	8.4	8.29
xmlGraphics	87.9	86.8	87.9	87.2	87.6	81.1	55.3	4	4	4	2	4	6.3	7.3
xmlSecurity	41.4	44.1	41.9	41.9	44.3	55.4	50.4	10	3	8	8	5	7.7	6.4

MORE+ significantly revealed more faults than those test suites reduced by applying most of the baseline approaches (see descriptive statistics in Table 4 and the results of the Mann-Whitney test as well as of the Benjamini-Hochberg correction). The unique exception is 2OPT that achieves results not statistically different from those obtained by applying *MORE+*. However, we can observe that the median values of RF for *MORE+* is 46.6%, for 2OPT is 52.5%, while the average of median values for HGS, DGR, GRD, *NSGAI12d* and GR3D is 56.7%. Table 5 (top-right) details the (mean) values of RF obtained for each application and considered suite reduction approach. We see that *MORE+* achieved the lowest RF on 7 out of 20 applications, while *NSGAI12d* on 5 out of 20 applications. Among the traditional approaches, 2OPT achieved better results. In detail, 2OPT achieved the lowest RF on 5 applications, DGR and GR3D respectively on 4 and 3 applications, and, finally, GRD and HGS on 2 applications. The results shown in Table 5 also suggest that all traditional approaches obtained comparable results for two applications (i.e., CommonsBcel and Jtidy), while for 14 applications at least three of these approaches obtained comparable results.

5.3 Reduction in test-suite execution time

The Kruskal-Wallis test returned 0.0036 as the p-value for RT (see Table 4), thus rejecting the null hypothesis NH_{RT} . For the pairs: *MORE+* - *NSGAI12d* and *MORE+* - GR3D, no statistical difference was observed (the Mann-Whitney test returned 0.338 and 0.046 as p-values, respectively). The time to execute suites reduced by *MORE+* was greater than the time needed to execute suites reduced by HGS, DGR, GRD, 2OPT. The median value of RT is 61.4% for our approach, while the (average) median value for these approaches is 73.9%. Results are confirmed by the \hat{A}_{12} test (>0.3) in favor of traditional approaches. This result is mainly related to the larger number of test cases in the test suites reduced by the evolutionary approach. That is, the larger the test suite (see results about the reduction in test-suite size), the greater the time to execute that suite is. Table 5 (down-left) shows the (mean) values of RT obtained on each application and the studied test suite reduction approaches.

5.4 Diversity

In Table 6, we show descriptive statistics for Div. We can observe that 36% is the (average) median value of test cases shared between test suites reduced by *MORE+* and

those reduced by the baseline approaches. This result might suggest that by considering more dimensions at the same time, we can obtain substantially different reduced test suites with respect to those obtained with traditional reduction approaches. Furthermore, GR3D is the approach that identified test suites having the highest percentage of test cases in common with suites produced by *MORE+*. In fact, *MORE+* and GR3D share in average 47.2% of test cases in contrast to 36.4% shared between *MORE+* and *NSGAII_{2d}* and 29.5% (in average) of test cases shared between *MORE+* and the other traditional reduction approaches. This outcome confirms that *MORE+* and both GR3D and *NSGAII_{2d}* have commonalities (respectively related to the considered objectives and algorithm), but also it suggests that the differences between these three approaches might have an impact on the reduced test suites.

5.5 Artifact coverage

In Table 7, we report the results of the Kruskal-Wallis test on the coverage of both code and requirements. Results suggest that no relevant difference existed in the observed coverage measures. In Tables 7 and 8, we report descriptive statistics (the coverage expressed in percentage) and raw data values (expressed in terms of lines of code and number of covered requirements) of the variables to estimate the coverage of both source code and application requirements. We can see some trend for which *MORE+* generated suites less effective in terms of code coverage than traditional approaches, while it achieved comparable results with respect to *NSGAII_{2d}*. This outcome was expected because *MORE+* and *NSGAII_{2d}* are based on evolutionary algorithms that balance among different objectives. While *MORE+* achieved a better requirements coverage with respect to traditional approaches, it was again comparable to *NSGAII_{2d}*. For instance, for *MORE+*, the mean values of Code_Cov is 88% and Reqs_Cov is 96.8% (see Table 7), for *NSGAII_{2d}* it was respectively 88.9% and 95.1%, while the (average) mean values of Code_Cov and Reqs_Cov for other traditional approaches were 98.7% and 90.6% respectively.

As a further analysis, for each application object of the study, we compared both code and requirement coverage of test suites reduced by *MORE+*, *NSGAII_{2d}*, and traditional approaches and their test effectiveness (e.g., “Are test suites with high code/requirements coverage more effective in discovering faults?”). To this aim, we used the Chi-squared test of independence to investigate the existing independence between code/requirements coverage and number of faults discovered. The conducted Chi-squared tests let us obtain always $p\text{-value} > 0.05$.

By a further manual inspection, we observed that for 17 out of 20 applications (85%), suites reduced by *MORE+* covered less code than traditional approaches, but in 12 of such applications (70.5%) *MORE+* discovered more faults than traditional approaches. Similarly, we observed that on 8 out of 20 applications (40%) *MORE+* covered more requirements than traditional approaches and

TABLE 6
Summary of statistics for Div

	Min	Median	Mean	Max	sd
<i>MORE+</i> / ZOPT	4.6	34	30.1	60	17.2
<i>MORE+</i> / DGF	0	34.3	30.8	60	17.1
<i>MORE+</i> / GRD	0	34	28.4	56.8	17.1
<i>MORE+</i> / HGS	0	33.3	28.5	58.4	17.9
<i>MORE+</i> / <i>NSGAII_{2d}</i>	13.2	33.5	36.4	75.5	15
<i>MORE+</i> / GR3D	22.4	46.9	47.2	100	16

TABLE 7
Summary of statistics for Code_Cov and Reqs_Cov (percentage)

	Min	Median	Mean	Max	sd
Code Cov (%)					
<i>MORE+</i>	47.5	91.1	88.0	100	11.3
ZOPT	74.3	100	98.7	100	5.7
DGF	74.3	100	98.7	100	5.7
GRD	74.3	100	98.7	100	5.7
HGS	74.3	100	98.7	100	5.7
<i>NSGAII_{2d}</i>	38.9	91.4	88.9	100	11.8
GR3D	74.3	100	98.7	100	5.6
Kruskal-Wallis ($p\text{-value} = 0.69$)					
Reqs Cov (%)					
<i>MORE+</i>	65.2	100	96.8	100	7.3
ZOPT	16.6	100	91.1	100	19.7
DGF	0.0	100	90.0	100	23
GRD	0.0	100	90.3	100	23.1
HGS	0.0	100	91.4	100	23
<i>NSGAII_{2d}</i>	66.6	100	95.1	100	7.5
GR3D	0.0	100	90.3	100	23.1
Kruskal-Wallis ($p\text{-value} = 0.706$)					

that in 7 of such applications (87.5%) it allowed us to discover more faults. With respect to suites reduced by *NSGAII_{2d}*, the suites reduced by *MORE+* covered slightly less code on 13 out of 20 applications (65%) and on 8 of these applications (61.5%) it discovered more faults. Concerning the requirements, *MORE+* covered slightly more requirements on 11 out of 20 applications (55%) and on 7 of them (63.6%) it discovered more faults. Even if these manual inspections were not supported by statistical evidences (e.g., for what concerns *NSGAII_{2d}* and *MORE+*), results suggest the mentioned trend in the gathered data.

If we consider these outcomes (together with the ones about RS and RF) related to code coverage (see Table 7), we can argue that they are not surprising since, as highlighted recently by Inozemtseva et al. [15], code coverage is not always directly correlated to fault detection capability. We can speculate that a similar observation could be drawn for the coverage of requirements. By weighting both code and requirements used to compute the artifact coverage, even when a less coverage degree was achieved, *MORE+* was inclined to focus the suite reduction on those parts of code and requirements that were more fault-prone, i.e., in which the probability of finding fault was higher. Moreover, by explicitly considering different types of information (low- and high-level such as code coverage/cost and requirements, respectively) when reducing a suite, *MORE+* reduced test suites without focusing on a dominant information (e.g., low-level such as code coverage/cost).

5.6 Cost of the reduction approach

In Tables 9 and 10, we report descriptive statistics and values for Cost_Red (i.e., the execution time of *MORE+* and

TABLE 8
Code and Requirement coverage values per application

APP	2OPT	DGR	GRD	GR3D	HGS	MORE+ (mean)	NSGAII _{2d} (mean)	2OPT	DGR	GRD	GR3D	HGS	MORE+ (mean)	NSGAII _{2d} (mean)
	Code Cov (Lines of code)							Reqs Cov (number of reqs.)						
AveCalc	426	426	426	426	426	424.9	424.8	9	9	9	9	9	9	9
CommonsBeel	2471	2471	2471	2471	2471	2685.1	2106.2	19	19	19	19	19	19.8	18.8
CommonsBeanUtils	3948	3948	3948	3948	3948	3506.3	3520.2	24	24	24	24	24	23.5	21.8
CommonsCodec	2539	2539	2539	2539	2539	2504.2	2507.2	15	15	15	15	15	16.9	17
CommonsCollections4	6477	6477	6477	6477	6477	4873	4987.5	16	16	16	16	16	15.9	15.8
CommonsIO	4098	4098	4098	4098	4098	3189	3526.2	18	18	18	18	18	18	18
CommonsLang	11186	11186	11186	11186	11186	10875.9	11075	16	15	16	16	16	16	16
CommonsProxy	598	598	598	598	598	534.7	534.6	9	9	9	9	9	9	8
DbUtils	749	749	749	749	749	737.6	716.8	7	7	7	7	9	8.7	8.2
iTrust	1951	1951	1951	1951	1951	1178.5	1536	14	14	14	14	14	14	14
Jabref	11004	11004	11004	11004	11004	10605	9372.4	24	24	24	24	24	24	23.4
Jtidy	259	259	259	259	259	250.1	252	15	15	15	15	15	17.24	21
JXPath	4755	4755	4755	4755	4755	3882.6	4010.9	19	19	19	19	19	20	20
LaTazza	69	69	69	69	69	69	69	1	0	0	0	0	6	6
Log4j	4053	4053	4053	4053	4053	2840.5	3581.2	23	23	23	23	23	21.2	17
Pmd	7529	7529	7529	7529	7529	6944	7058	18	18	18	18	18	18	17.7
Woden	2223	2223	2223	2223	2223	2099.7	1958.3	20	20	20	20	20	20.9	20.4
Xerces	12676	12676	12676	12676	12676	12410.5	12463.6	20	20	20	20	20	20	20
xmlGraphics	4230	4230	4230	4230	4230	3360.7	3846.9	19	19	19	19	19	17.9	17.2
xmlSecurity	2830	2830	2830	2830	2830	2531.4	2682.4	14	14	14	14	14	13.4	12.8

TABLE 9
Summary of statistics for *Cost_Red*

	Algorithm Execution Time (in sec ds)				
	Min	Median	Mean	Max	sd
MORE+	0.0003	0.04	0.08	0.51	0.12
2OPT	0.05	3	48	720.5	159.8
DGF	0.003	0.17	0.25	0.76	0.22
GRD	0.009	0.41	0.77	3.38	0.8
HGS	0.0025	0.03	0.039	0.11	0.03
NSGAII _{2d}	0.0003	0.04	0.09	0.8	0.18
GR3D	0.0005	0.33	0.94	5.2	1.33
Similarity Links Recovering (in seconds)					
	Reqs - TestCases		Reqs - Code		
Min	2		10		
Median	82.5		19		
Mean	165.6		50.9		
Max	1505		430		
sd	328.6		93.7		

baselines). These values indicate the time to obtain S_{red} , expressed in seconds. Results suggest that on average *Cost_Red* varied from a few seconds to a few minutes. 2OPT required more time than other approaches. We can assume that this was due to fact that 2OPT is a variant of the greedy approach that conduces all-pairs comparison of test cases to reduce test suite, i.e., it performs a high number of comparison to reduce a test suite. We also observed that in some cases (e.g., CommonsLang, Pmd, and CommonsIO), 2OPT required an execution time notably greater than other approaches (more than 30 seconds for 2OPT vs. less than 3 seconds for other approaches). However, MORE+ and GR3D need information extracted by means of textual-based similarity links to reduce test suites. In Table 9, we also report the time (in seconds) to recover these links. In detail, the table reports the time to recover similarity links between: (i) requirements and test cases and (ii) requirements and source code. The first case was more costly since it required on average 165.6 seconds. The recovery of links between requirements and source code on average required 50.9 seconds. We argue that this finding was due to the granularity of the analysis conducted to recover similarity links: methods of JUnit classes (i.e., test cases) and Java classes, respectively. Overall, MORE+ requires more time than other approaches,

depending on the analyzed application, the additional time required ranges from 0.2 seconds to 14 minutes while on average the additional time corresponds to 2.7 seconds. The recovery of similarity links could require a non-trivial amount of time, but we can postulate that the time to recover links might be hidden to an end-user given that the recovery process is executed in background only when either requirements or source code are added/modified. To further study the cost of the reduction approach, we measured the relationship between cost and effectiveness, by means of CostEff (see next subsection).

5.7 Cost-Effectiveness

To compute CostEff values, we considered 3 subsequent versions of each experimental object (i.e., $g = 3$ in Equations (15) and (16)). Table 11 reports descriptive statistics for CostEff. We can observe that: (i) MORE+ was overall competitive even if it was applied to only one version of the application to-be-tested, in fact, the value of CostEff for MORE+ was worse than DGR while it outperformed all the other approaches; (ii) an increase in the number of consecutive application releases showed an increase in competitiveness (lower cost) of MORE+ with respect to other approaches and more traditional ones. In other terms, an increased number of application versions corresponded to a growing competitiveness of MORE+ (i.e., lower cost and high effectiveness with respect to other techniques), while 2OPT, GRD and HGS decreased their competitiveness in terms of CostEff (i.e., higher cost and less effectiveness with respect to other techniques). It is worth mentioning that GR3D was always worse than other approaches. It was due to the fact that it was a quite simple and heavy algorithm (greedy) that required a textual analysis (link recovery) among software artifacts.

6 ADDITIONAL ANALYSIS

In the following, we present the results of additional analyses on the gathered data. In particular, we present the results of an analysis on the impact of the size of reduced

TABLE 10
Cost_Red values for experimental objects

APP	Algorithm Execution Time (second)							Similarity Links Recovering (second)		Total cost	
	MORE+ (mean)	GRD	2OPT	DGR	HGS	NSGAI1 _{2d} (mean)	GR3D	Reqs - TestCases	Reqs - Code	GR3D	MORE+ (mean)
AveCalc	0.002	0.08	0.05	0.03	0.003	0.002	0.09	2	10	2.09	12
CommonsBcel	0.02	0.15	0.44	0.09	0.02	0.02	0.11	27	25	27.1	52
CommonsBeanUtils	0.25	2.18	36.09	0.76	0.03	0.17	3.24	220	10	223.2	230.5
CommonsCodec	0.04	0.18	6.25	0.13	0.02	0.04	0.26	75	10	75.2	85
CommonsCollections4	0.07	0.81	30.93	0.38	0.03	0.09	2.23	160	80	162.2	240
CommonsIO	0.09	0.83	55.11	0.34	0.04	0.08	0.81	130	30	130.8	160
CommonsLang	0.52	3.38	720.50	0.72	0.12	0.80	5.27	1505	74	1510.2	1579.5
CommonsProxy	0.00	0.12	0.07	0.04	0.06	0.002	0.01	6	28	6	34
DbUtils	0.01	0.14	0.50	0.12	0.01	0.01	0.06	13	10	13	23
iTrust	0.05	0.26	7.55	0.15	0.01	0.05	0.26	120	11	120	131
Jabref	0.07	0.77	3.64	0.45	0.04	0.07	1.03	200	104	201	304
Jtidy	0.01	0.05	0.23	0.02	0.01	0.01	0.01	20	13	20	33
JXPath	0.05	0.48	1.81	0.20	0.03	0.05	0.40	36	35	36.4	71
LaTazza	0.0003	0.0092	0.1537	0.0030	0.0026	0.0003	0.0005	3	10	3	13
Log4j	0.07	1.77	9.34	0.50	0.04	0.07	1.20	90	12	91.2	102
Pmd	0.20	1.81	91.34	0.42	0.06	0.30	2.05	300	75	302	375.2
Woden	0.04	1.14	2.39	0.20	0.02	0.04	0.63	90	10	90	100
Xerces	0.17	0.77	6.79	0.31	0.05	0.16	0.75	273	430	273.7	703.1
xmlGraphics	0.02	0.27	0.90	0.11	0.02	0.02	0.26	27	10	27.2	37
xmlSecurity	0.02	0.36	1.70	0.11	0.10	0.02	0.24	15	31	15.2	46

TABLE 11
CostEff: statistics on the 20 applications

App.# Release	MORE+	2OPT	DGR	GRD	HGS	NSGAI1 _{2d}	GR3D
Mean CostEff							
1	1254	1262	1214	1268	1261	1274	1446
2	2291	2475	2427	2535	2522	2331	2725
3	3328	3689	3640	3802	3783	3388	4005
Std.dev CostEff							
1	689.6	641.4	62.5	539.7	458.0	674.1	788.8
2	1111.1	1182.1	925.1	1079.1	915.9	1089.2	1342.7
3	1551.7	1727.1	1387.6	1618.6	1373.8	1525.2	1910.5

test suites and their capability in discovering faults. We also show the results of an analysis on the Pareto fronts built by *MORE+* and conclude presenting the results of an analysis of co-factors on the main results of our study.

6.1 The impact of suite size in fault discovery

To additionally investigate the impact of the size (number of test cases) of test suites on test effectiveness (capability of discovery faults), we first compared the results achieved by *MORE+* with the ones obtained by randomly reducing test suites (fixing the size of reduced suites according to the ones generated by *MORE+*). In other terms, for each experimental object, we applied a random reduction (named Rand) of its test suite and measured the number of faults discovered by such reduced suites. We considered the average of 20 random reductions for each application. Table 12 details results for the comparison: Rand vs *MORE+*/traditional approaches (e.g., 2OPT, GRD), as well as descriptive statistics and statistical tests. By comparing Table 12 with results shown in Table 4 and Table 5, we can observe that even if the randomly reduced suites had the same size of the suites reduced by *MORE+*, they had a lower capability to find faults and this was statistically confirmed by the Mann-Whitney test (p -value <0.001) and by \hat{A}_{12} (>0.19). The mean difference was more than 4.2 (out of 17.9 on average) faults additionally discovered per application by *MORE+*. Similarly, even if the randomly reduced suites were larger than the suites reduced by

traditional approaches, they were not more effective in terms of faults discovery.

Furthermore, we compared the suites reduced by *MORE+* and traditional approaches according to their size and test effectiveness (e.g., “Are large suite more effective in discovering faults than small ones?”). To this aim, we adopted the one-way permutation test lets us identify any interaction (impact) between two considered variables; in our case: suite size and their effectiveness. While the one-way permutation test computed for the traditional approaches returned 0.02 as p -value, i.e., showing the existence of a given impact of suite size in suite effectiveness. Conversely, the one-way permutation test computed for the *MORE+* returned 0.26 as p -value, thus it does not evidence any impact of suite size in suite effectiveness. By comparing the size of reduced suites obtained by traditional approaches with the ones of *MORE+* and their test effectiveness (see Table 13), we observed that, on one side, for 11 out of 20 applications *MORE+* generated larger suites with respect to the ones generated by traditional approaches. However, only in 6 cases, out of these 11 (55%), such larger suites were actually more effective in finding faults than the suites reduced by traditional approaches. On the other side, we also observed that in the other 9 applications, *MORE+* generated suites smaller than their counterpart generated by traditional approaches. Nevertheless their smaller size, in 7 out of such 9 applications (77%), these suites were more effective in finding faults.

Overall, the results of this analysis suggested that there is still room to improve *MORE+* with respect to the effective of the results, in fact, in a few cases we observed that traditional approaches competitively reduced test suites. However, results also clearly indicated that the suite size did not play a fundamental role on test effectiveness.

TABLE 12

Faults: descriptive statistics and the mean values per application for Random and statistical tests (in bold values significant at 5%, while * indicates values also significant by applying the Benjamini-Hochberg correction)

Fault					
	Min	Median	Mean	Max	sd
Random	0	4	5.2	12	17.2
Fault mean for application					
App	Fault mean	App	Fault mean		
AveCalc	8.7	Jabref	9.3		
CommonsBcel	3	Jtidy	0.58		
CommonsBeanUtils	8.2	JXPath	3.5		
CommonsCodec	8	LaTazza	7.3		
CommonsCollections4	5.6	Log4j	3.3		
CommonsIO	4.5	Pmd	3.4		
CommonsLang	3.5	Woden	9.8		
CommonsProxy	4.6	Xerces	3		
DbUtils	6	xmlGraphics	2.6		
iTrust	6.4	xmlSecurity	3		
Kruskal-Wallis p-value < 0.001					
			Mann-Whitney p - value	\hat{A}_{12}	
Rand - MORE+			0.001*	0.19	
Rand - 2OPT			0.012*	0.32	
Rand - DGR			0.012*	0.32	
Rand - GRD			0.047	0.35	
Rand - HGS			0.025*	0.34	
Rand - GR3D			0.074	0.37	

TABLE 13

F (Faults) vs Suite size: comparison between traditional approaches, MORE+ and NSGAI_{2d}

	Suite Size (mean)			Faults (mean)		
	Traditional	MORE+	NSGAI _{2d}	Traditional	MORE+	NSGAI _{2d}
AveCalc	6	10	10	10.2	12.9	11.6
CommonsBcel	46.6	29	27	7	8.6	6.8
CommonsBeanUtils	234.2	408	408	14.8	14.6	14
CommonsCodec	95.8	216	216	2.4	10.4	10
CommonsCollections4	322.8	319	319	11.4	9.4	11.3
CommonsIO	389.6	343	429	9.2	10.3	11.2
CommonsLang	96.8	385	577	2.4	7.8	9.7
CommonsProxy	38.4	35.8	30	6.6	5.6	7.3
DbUtils	49.4	128	80	7.2	12.5	9.5
iTrust	170	259	433	6.2	8.4	12.4
Jabref	91	125	62	10.8	13	5.7
Jtidy	29	28	144	3	4.4	7.7
JXPath	73	66	66	8.2	8.5	8.6
LaTazza	1.2	8.8	7	1.6	9.8	9
Log4j	285	93.5	236	18.6	10	16.8
Pmd	100.4	209	206	6.0	6.2	5.2
Woden	107.8	177	99	10.2	14.7	7.9
Xerces	24	57	57	7.2	8.4	8.2
xmlGraphics	50.4	50	84	3.6	6.3	7.3
xmlSecurity	36	36	36	6.8	7.7	6.4

6.2 Analysis of Pareto fronts

To study the Pareto fronts built by MORE+, we plotted: the mean effectiveness values of traditional approaches (the black line in Figure 4), and the number of faults discovered by each suite per experimental object for both MORE+ (see the blue boxplots in Figure 4) and NSGAI_{2d} (see the white boxplots in Figure 4). We recall that Table 13 reports the mean values of discovered faults for MORE+, NSGAI_{2d}, and traditional approaches.

By looking at the boxplots and the (mean) trends in Figure 4, we can identify the following groups of applications (experimental objects):

- Group (A): applications for which almost all suites (more than 90%) in the Pareto fronts were more effective than the mean trend of traditional approaches:

9 for MORE+ and 7 for NSGAI_{2d}.

- Group (B): applications for which a large amount (between 65% and 90%) of suites in the fronts were more effective than the mean trend of traditional approaches: 5 for MORE+ and 4 for NSGAI_{2d}.
- Group (C): applications for which in a range between 50% and 65% of suites in the fronts were more effective than the mean trend of traditional approaches: 1 for MORE+ and 2 for NSGAI_{2d}.
- Group (D): applications for which almost all (more than 90%) of suites in the fronts were less effective than the mean trend of traditional approaches: 1 for MORE+ and 2 for NSGAI_{2d}.
- Group (E): applications for which a large amount (between 65% and 90%) of suites in the fronts were less effective than the mean trend of traditional approaches: 3 for MORE+ and 2 for NSGAI_{2d}.
- Group (F): applications for which in a range between 50% and 65% of suites in the fronts were less effective than the mean trend of traditional approaches: 1 for MORE+ and 3 for NSGAI_{2d}.

These groups indicated that on 18 out of 20 applications for MORE+ and 15 out of 20 for NSGAI_{2d} (those in the groups A, B, D, E), the mean trend of test effectiveness of all suite in the Pareto fronts was representative of the effectiveness of the reduced suites that compose such fronts. For instance, we can observe that almost all suites of Pareto fronts of MORE+ and NSGAI_{2d} related to the 9 and 7 applications of Group (A) respectively were superior, in terms of test effectiveness, than those suites reduced with traditional approaches. Hence, we can deduce that studying the average test effectiveness of these fronts is enough representative for all their suites. This did not hold for the 2 and 5 applications of Groups (C) and (F) of MORE+ and NSGAI_{2d} respectively, for which the mean test effectiveness of suites in the fronts seemed to be not enough representative, in fact, comparing different reduced suites of the fronts with the mean effectiveness of traditional approaches could led to different results.

Furthermore, boxplots for MORE+ and NSGAI_{2d} in Figure 4 graphically expressed the statistical data already shown in Table 4. Such data suggest that MORE+ discovered significantly more faults than NSGAI_{2d} in several of the considered experimental objects.

As a further analysis, Table 14 reports the difference, in terms of average number of discovered faults (Eff.), between suites in the Pareto fronts and the mean effectiveness of traditional approaches per each group of experimental objects for MORE+ and for NSGAI_{2d}. Focusing on the upper part of Table 14 related to MORE+, we observed, for instance, a clear trend in Group (A): by considering 252 reduced suites that had higher effectiveness than the suites reduced by traditional approaches (i.e., “Eff(MORE+) > Eff(Trad mean)” in Table 14), out of 255 suites in the Pareto fronts produced by MORE+ (in group A), the mean number of faults additionally discovered by MORE+ was 4.59 (row 3 in Table 14). Conversely, only for a few suites (3 out of 255), in the fronts for which traditional approaches

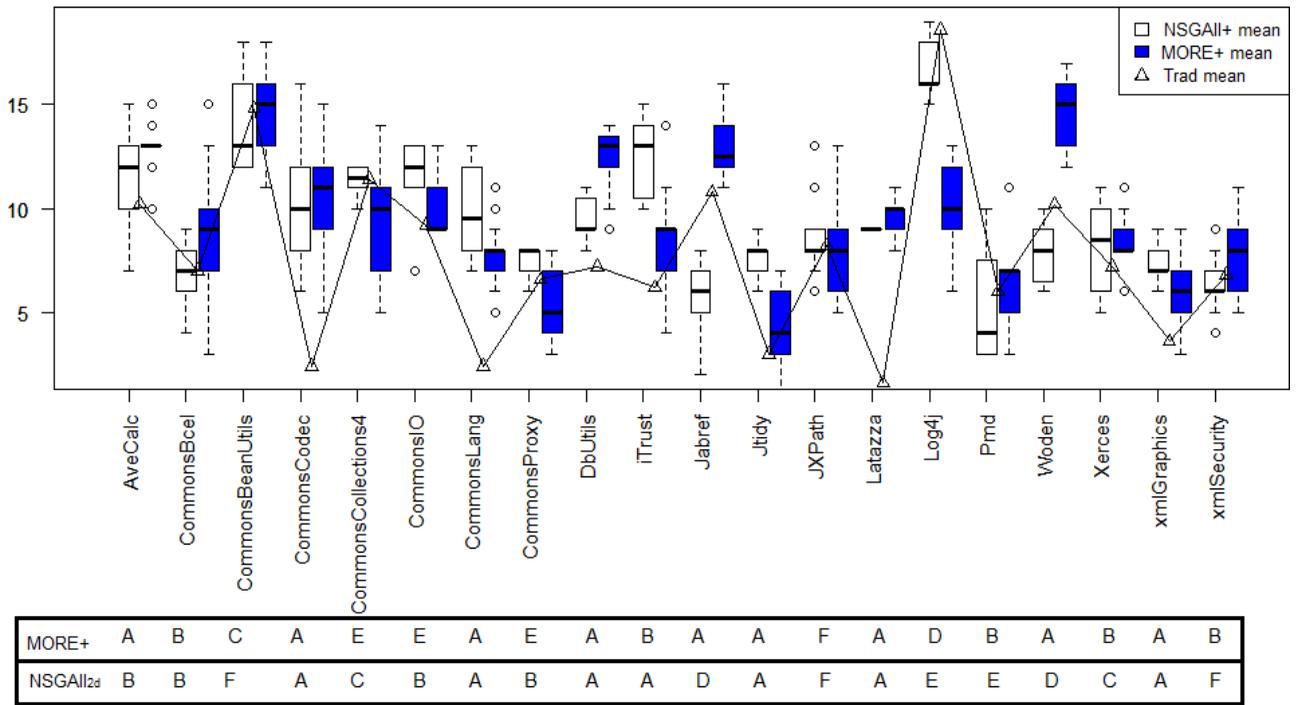


Fig. 4. Box-plots of faults discovered by all solutions in Pareto fronts of *MORE+* (blue filled boxplots) and *NSGAI_{2d}* (white boxplots), mean effectiveness of traditional approaches (continuous black line), and their groups (letters from A to F) for *MORE+* and *NSGAI_{2d}*

achieved better results (i.e., $\text{Eff}(MORE+) < \text{Eff}(\text{Trad mean})$ in Table 14), the mean number of faults additionally discovered was 0.93 (row 4 in Table 14). In the groups (C) and (F), we could note contrasting results. Looking at group (C), we observed that: (i) by considering those suites reduced by *MORE+* that had higher effectiveness than suites reduced by traditional approaches (24 out of 46 of Pareto fronts), the mean number of faults additionally discovered by *MORE+* was 1.26 (row 7 of Table 14); and (ii) by considering those suites reduced by *MORE+* that had achieved less effectiveness than the ones reduced by traditional approaches (22 out of 46 of Pareto fronts), the mean of faults additionally discovered was 1.71 (row 8 of Table 14). Concerning the group (F), we observed that: (i) by considering those suites reduced by *MORE+* that had higher effectiveness than the suites reduced by traditional approaches (10 out of 22 of Pareto fronts), the mean number of faults additionally discovered by *MORE+* was 1.5 (row 13 of Table 14); and (ii) by considering those suites reduced by *MORE+* that had achieved less effectiveness than the ones reduced by traditional approaches (12 out of 22 of Pareto fronts), the mean of faults additionally discovered was 1.53 (row num. 14 of Table 14).

The lower part of Table 14 show the results of the same analysis on Pareto fronts produced by *NSGAI_{2d}*. Similar considerations to those reported for *MORE+* can be done. Also in this case Groups (C) and (F) provided contrasting results. For example, in the group (C) we observed that: (i) by looking at the 12 out of 20 suites reduced by *NSGAI_{2d}* they had higher effectiveness than the suites

TABLE 14

Faults: Test effectiveness difference between suites in the Pareto fronts and the ones reduced by traditional approaches

Group	Eff	Nr. suites ∈ Pareto fronts	Min	Median	Mean	Max	sd
MORE+							
A	$\text{Eff}(MORE+) > \text{Eff}(\text{Trad mean})$	252	0	4.6	4.59	12.6	2.67
A	$\text{Eff}(MORE+) < \text{Eff}(\text{Trad mean})$	3	0.2	0.6	0.93	2	0.95
B	$\text{Eff}(MORE+) > \text{Eff}(\text{Trad mean})$	112	0	1.8	1.94	8	1.56
B	$\text{Eff}(MORE+) < \text{Eff}(\text{Trad mean})$	29	0.2	1	1.26	4	0.78
C	$\text{Eff}(MORE+) > \text{Eff}(\text{Trad mean})$	24	0.2	1.2	1.2	3.2	0.88
C	$\text{Eff}(MORE+) < \text{Eff}(\text{Trad mean})$	22	0.8	1.8	1.71	3.8	0.97
D	$\text{Eff}(MORE+) > \text{Eff}(\text{Trad mean})$	0	-	-	-	-	-
D	$\text{Eff}(MORE+) < \text{Eff}(\text{Trad mean})$	24	0.2	2.4	2.47	6.4	1.67
E	$\text{Eff}(MORE+) > \text{Eff}(\text{Trad mean})$	19	0.4	1.4	1.2	3.8	0.86
E	$\text{Eff}(MORE+) < \text{Eff}(\text{Trad mean})$	49	0.2	2.4	2.47	6.4	1.67
F	$\text{Eff}(MORE+) > \text{Eff}(\text{Trad mean})$	10	0.8	0.8	1.5	4.8	1.25
F	$\text{Eff}(MORE+) < \text{Eff}(\text{Trad mean})$	12	0.2	2.2	1.53	3.2	1.07
NSGAI_{2d}							
A	$\text{Eff}(MORE+) > \text{Eff}(\text{Trad mean})$	58	0.8	5	5.51	13.6	2.59
A	$\text{Eff}(MORE+) < \text{Eff}(\text{Trad mean})$	0	-	-	-	-	-
B	$\text{Eff}(MORE+) > \text{Eff}(\text{Trad mean})$	27	0	1.8	1.79	4.8	1.47
B	$\text{Eff}(MORE+) < \text{Eff}(\text{Trad mean})$	12	0.2	1.6	1.63	3.2	0.92
C	$\text{Eff}(MORE+) > \text{Eff}(\text{Trad mean})$	12	0.6	2.3	1.91	3.8	1.17
C	$\text{Eff}(MORE+) < \text{Eff}(\text{Trad mean})$	8	0.2	1.2	1.03	2.2	0.66
D	$\text{Eff}(MORE+) > \text{Eff}(\text{Trad mean})$	0	-	-	-	-	-
D	$\text{Eff}(MORE+) < \text{Eff}(\text{Trad mean})$	33	0.2	3.8	4.04	8.8	2.12
E	$\text{Eff}(MORE+) > \text{Eff}(\text{Trad mean})$	2	0.4	2.2	2.2	4	2.55
E	$\text{Eff}(MORE+) < \text{Eff}(\text{Trad mean})$	9	0.6	2.6	2.19	3.6	1.13
F	$\text{Eff}(MORE+) > \text{Eff}(\text{Trad mean})$	18	0.2	0.8	1.21	4.8	1.27
F	$\text{Eff}(MORE+) < \text{Eff}(\text{Trad mean})$	23	0.2	0.8	1.24	2.8	0.92

reduced by traditional approaches, the mean number of faults additionally discovered by *NSGAI_{2d}* was 1.91 (row 20 of Table 14); and (ii) by considering those suites reduced by *NSGAI_{2d}* that have achieved less effectiveness than those reduced by traditional approaches (8 out of 20 of the Pareto fronts), the mean of faults additionally discovered was 1.03 (row 21 in Table 14). A similar analysis could be performed in the group (F).

Furthermore, to compare all the suites reduced by *MORE+* and those reduced by traditional approaches, we

built and compared two Pareto fronts for each application: the first one was composed of all suites reduced by *MORE+* and the second one, namely *reference Pareto front*, composed by the suites reduced by *MORE+* and by the traditional approaches. As described in the work of Yoo et al. [16], taking advance of the Pareto optimality, a *reference Pareto front* is a front specifically constructed and used when comparing solutions produced by different algorithms according to the dimensions of the reference front itself. Specifically, among all the solutions from the union of the different Pareto fronts (those from *MORE+* and from the traditional approaches), only the non-dominated ones, i.e., those solutions that are better than the others according to the considered dimensions, were preserved and became part of the final *reference Pareto front*. The construction of the *reference Pareto front*, hence, helps in identifying the best (sub)set of solutions, according to the adopted dimensions. In this work, we built the Pareto fronts by considering the “number of discovered faults”, “code coverage” and “execution cost” of each suite as dimensions. The reason behind the choice of these dimensions was that they are often adopted to estimate the effectiveness of test suites.

By comparing such reference Pareto fronts per application, we could observe that suites reduced by *MORE+* were not dominated (Mann-Whitney p-value = 1), instead, suites reduced by traditional approaches were only partially not dominated (p-value < 0.01). In fact, in the reference fronts, on average, 100% of suites reduced by *MORE+* were not dominated by suites reduced by traditional approaches (for all 20 applications, 100% of suites reduced by *MORE+* were not dominated). Conversely, only 26% of suites reduced by traditional approaches were not dominated. We measured the Hypervolume (HV) metric [78], [63] of each reference front and compared them by applying the Mann-Whitney test and the \hat{A}_{12} test (see Table 15). HV is a metric that computes the volume (i.e., size) covered by solutions in a front obtained by considering an objective space. The hypervolume is hence the space that is contained by all solutions of a front with respect to a given reference point, computed by considering the maximum possible objective values. HV is a quality indicator for evaluating and comparing Pareto fronts that measures the convergence and uniform diversity of the front, i.e., a larger hypervolume is typical for Pareto fronts having a better trade-offs among the considered objectives than solutions having a smaller hypervolume. On the obtained HV values, we computed the Mann-Whitney test (p-value = 0.08) and \hat{A}_{12} (\hat{A}_{12} = 0.34) for evaluating the trend of all the experimental objects. Results suggest that by additionally considering suites generated by traditional approaches, no substantial improvements were obtained in the set of suites reduced by *MORE+*. This seems to confirm once again that *MORE+* is promising in reducing test suites.

Additionally, we compared all the suites reduced by *NSGAII_{2d}* and those reduced by traditional approaches, we built and compared two Pareto fronts for each experimental object: the first one was composed of all the suites reduced by *NSGAII_{2d}* and the *reference Pareto*

front composed by the suites reduced by *NSGAII_{2d}* and by the traditional approaches. By comparing such reference Pareto fronts per application, we observed that suites reduced by *NSGAII_{2d}* were mainly not dominated (Mann-Whitney p-value = 1), instead, suites reduced by traditional approaches were only partially not dominated (p-value < 0.01). In fact, in the reference fronts, on average, 100% of suites reduced by *NSGAII_{2d}* were not dominated by suites reduced by traditional approaches (for all 20 applications, 100% of suites reduced by *NSGAII_{2d}* were not dominated). Conversely, only 28% of suites reduced by traditional approaches were not dominated. On the obtained HV values (see Table 15), we computed the Mann-Whitney test that returned 0.01 as the p-value. The \hat{A}_{12} value was 0.75. Results suggests that by additionally considering suites generated by traditional approaches, no substantial improvements were obtained in the set of suites reduced by *NSGAII_{2d}*. This outcome seems to suggest that *NSGAII_{2d}* achieved good results in reducing test suites.

Finally, we compared all the suites reduced by *MORE+* and *NSGAII_{2d}*, we built and compared two Pareto fronts for each application: the first one was composed of all the suites reduced by *MORE+* and the *reference Pareto front* composed by the suites reduced by *MORE+* and by *NSGAII_{2d}*. By comparing such reference Pareto fronts per application, we observed that the suites reduced by *MORE+* were mainly not dominated (Mann-Whitney p-value < 0.01), instead, the suites reduced by *NSGAII_{2d}* were only partially not dominated (p-value < 0.01). In fact, in the reference fronts, on average, 97.1% of suites reduced by *MORE+* were not dominated by suites reduced by *NSGAII_{2d}* (17 out of 20 applications, 100% of suites reduced by *MORE+* were not dominated). Conversely, only 66% of suites reduced by *NSGAII_{2d}* were not dominated (only for 8 applications, 100% of suites reduced by *NSGAII_{2d}* were not dominated). On the obtained HV values (see Table 15), we computed the Mann-Whitney test and \hat{A}_{12} . The obtained values were 0.05 and 0.6825, respectively. Results show that by additionally considering suites generated by *NSGAII_{2d}*, no substantial improvements were obtained in the set of suites reduced by *MORE+*. This seems to confirm once again that *MORE+* is promising in reducing test suites.

6.3 Analysis of Co-factors

In Table 16, we summarize the results concerning how the considered faults impact on the requirements. This table reports (first column) the percentage of requirements affected by at least one fault. For instance, 40% (i.e., 4 out of 10) of requirements considered for AveCalc were affected by at least one fault. Table 16 also reports (second column) the percentage of faults not impacting on any requirements. For instance, all considered faults affect application requirements. Moreover, this table reports (third column) the percentage of test cases that did not impact on requirements. For examples, 3 out of 20 faults (15%) of CommonsCollections did not impact on the set of considered requirements.

TABLE 15
HV data computed for the built reference Pareto fronts

App	$MORE^+$ vs HV_{MORE^+}	trad HV_{Call}	$NSGAI2d$ vs $HV_{NSGAI2d}$	trad HV_{Call}	$MORE^+$ vs N HV_{MORE^+}	$GAI2d$ vs HV_{Call}
AveCalc	0.0	0.0	0.2	0.2	7.3	0.0
CommonsBcel	2198.6	901.1	249.6	249.6	3827.9	901.1
CommonsBeanUtils	139976.7	139976.7	58.8	58.8	140045.0	139976.7
CommonsCodec	9648.7	131.6	2414.9	101.0	234.2	131.6
CommonsCollections4	36777.6	4467.4	2445.4	48.0	4467.4	4467.4
CommonsIO	8678.4	345.6	5710.0	0.0	8154.5	345.6
CommonsLang	179978.5	1113.3	14976.5	0.0	52878.3	1113.3
CommonsProxy	377.6	225.1	0.0	0.0	225.1	225.1
DbUtils	1092.1	30.4	807.1	1.9	499.0	30.4
iTrust	107.0	107.0	92016.1	121.0	12290.8	107.0
Jabref	153385.2	2527.6	2554.1	2554.1	106791.2	2527.6
JTidy	1.6	1.6	230.9	0.3	206.4	1.6
JXPath	7414.9	1321.7	1349.3	528.1	1415.8	1321.7
LaTazza	0.0	0.0	0.0	0.0	0.0	0.0
Log4j	208429.7	2838.3	2502.7	0.0	131177.1	2838.3
Pmd	2352.6	2352.6	481.7	481.7	2509.8	2352.6
Woden	7454.4	105.3	1220.6	39.2	4289.1	105.3
Xerces	810.2	810.2	196.9	1036.4	810.2	810.2
xmlGraphics	38490.1	16305.8	16133.5	2573.3	40681.6	16305.8
xmlSecurity	5708.9	4490.5	2042.1	558.6	44192.3	4490.5
μ - value	0.08		0.01		0.05	
A_{12}	0.34		0.75		0.68	

TABLE 16
Impact of faults on requirements

App	% Reqs affected by fault	% Faults not impacting reqs	% Not tested Req	Fault Density
LaTazza	60	13	40	1.8
AveCalc	40	0	10	3.2
CommonsProxy	80	0	10	1.7
DBUtils	58	14	25	1.6
iTrust	80	23	0	1.8
CommonsCodec	57	35	10	2
JTidy	36	6	8	1.7
Woden	41	21	0	1.8
Log4J	41	0	4	1.7
JXPath	55	15	0	2
CommonsIO	50	0	0	3.4
CommonsBcel	25	0	0	1.6
CommonsBeanUtils	57	22	7	1.8
xmlGraphics	50	20	12	4.3
xmlSecurity	13	0	0	1.9
CommonsCollections	41	15	5	1.8
Pmd	50	10	10	2
CommonsLang	80	5	0	2.2
Jabref	51	50	22	1.5
Xerces	80	0	0	1.6

In the fourth column of Table 16, we report fault density

$$(FaultDensity = \sum_{r \in Reqs} \left(\frac{NumFaults_r - \text{mean}(\frac{NumFaults}{NumReqs})}{NumReqs} \right)).$$

High values for fault density indicate applications with faults concentrated in a few requirements, while low levels of fault density indicate applications with faults spread among many requirements.

From Table 16, we can also see that faults were evenly distributed among application requirements (i.e., distributed in more than 51% of requirements — median value for Reqs affected by faults — and with a fault density lower or equal than 1.8 — median value for FaultDensity) in case of: LaTazza, CommonsProxy, DbUtils, iTrust, CommonsBeanUtils, and Xerces. Conversely, the faults were concentrated in a few requirements (i.e., distributed in less than 51% of requirements and with a fault density higher or equal than 1.8) in: AveCalc, Woden, CommonsIO, xmlSecurity, and CommonsCollections.

On the basis of the results shown in Table 16 (fourth column), a high number of requirements for LaTazza and DbUtils (4 out of 10 and 3 out of 12, respectively) were not linked to any test case. This result suggests that test cases were mainly focused on a subset of the considered

TABLE 17
Percentage of test cases revealing: at least one fault, more than one fault for each application; and the functional test case redundancy

App	TCS revealing ≥ 1 fault	TCS revealing > 1 fault	TCS revealing 1 fault	TCS Redundancy
LaTazza	68	51	17	8.2
AveCalc	53	49	4	11.7
CommonsProxy	8	7.8	7.8	5.6
DbUtils	8	1	7	10.5
iTrust	4	0.4	3.6	3.6
CommonsCodec	5	0.1	4.9	13.8
JTidy	4	0.3	3.7	14.4
Woden	8	0.8	7.2	4.2
Log4j	4	0.9	3.1	21
JXPath	7	0.4	6.6	6.5
CommonsIO	3	0.2	2.8	10.8
CommonsBcel	26	2	24	4.1
CommonsBeanUtils	3	0.2	2.8	15.8
xmlGraphics	8	0	8	3.3
xmlSecurity	18	0	18	3.2
CommonsCollections	3	0.2	2.8	4.3
Pmd	3	0.1	2.9	3.8
CommonsLang	1.4	0.3	1.1	20
Jabref	13	0	13	9.4
Xerces	7	0.7	6.3	8.5

requirements. As for iTrust, Woden, JXPath, CommonsIO, CommonsBcel, xmlSecurity, CommonsLang, and Xerces, all the requirements were linked to at least one test case, while for the other applications a few requirements (on average 7.7% for each application) were not linked with test cases, even if some links were present. This outcome suggests that the set of similarity links used in the empirical study could be incomplete.

Results reported in the second column of Table 17 suggest that test suites of AveCalc, LaTazza, CommonsBcel, xmlSecurity, and Jabref had a non-trivial percentage of test cases revealing at least one fault. Conversely, a large number of test suites (i.e., the ones of: iTrust, JTidy, Log4J, CommonsIO, CommonsBeanUtils, CommonsCollections, Pmd, CommonsLang) had less than 5% of fault-revealing test cases. For each application, the third column of Table 17 shows the percentage of test cases revealed more than one fault, respectively. We can observe that only in AveCalc and LaTazza a large percentage of test cases (about 50%) revealed more than one fault, while in other applications almost all test cases revealed only one fault. The fourth column of Table 17 reports the percentage of test cases that revealed only one fault, we can see that: in average less than 8% of test cases revealed only one fault, LaTazza had the highest percentage 17%, while 10 applications had less than 5% of test cases that revealed only one fault.

In the last column of Table 17, we report the results for test case redundancy (TCSRedundancy). This measure was computed as follows: $\frac{|TCS|}{|Testclasses|}$. TCS is the set of test cases composing a suite and Testclasses is the set of JUnit classes that functionally group test cases. We assume that JUnit classes group functionally correlated test cases, i.e., JUnit test methods. Results suggest that the test suites with high redundancy are those of: AveCalc, Log4j, CommonsLang, CommonsBeanUtils, JTidy, and CommonsCodec.

TABLE 18
Spearman’s correlation of co-factors. Bold values are statistically significant at 95%

Co-factor	RF	RS	RT
MORE+			
Reqs	-17%	-2%	-7%
NotTestedReqs	-19%	9%	-13%
TCSRedundancy	-11%	34%	10%
TCSRevealing \geq 1fault	-11%	-3%	-20%
TCSRevealing1fault	-4%	-5%	-22%
GR3D			
Reqs	-23	-22%	-28%
NotTestedReqs	-35	-3	-17
TCSRedundancy	-23	5	-15
TCSRevealing \geq 1fault	-33	-15	-30
TCSRevealing1fault	-23	-19	-33
Other approaches			
Reqs	15%	22%	15%
NotTestedReqs	-11%	23%	4%
TCSRedundancy	-18%	38%	11%
TCSRevealing \geq 1fault	4	27%	18%
TCSRevealing1fault	23%	21%	17%

The results of a two-way permutation test indicated that Reqs, NotTestedReqs, and TCSRevealing1fault had a statistically significant effect on: RF, RS and RT (p-value is always less than 0.001). A two-way permutation test showed that TCSRedundancy and TCSRevealing \geq 1fault had a statistically significant impact (p-value \leq 0.02) on RF/RT and RS, respectively. No co-factors had a significant effect on the interaction with studied reduction approaches/techniques.

The results of a two-way permutation test also suggested that the observed outcomes depend on the experimental objects: we obtained a p-value less than 0.001 for RF, RS, and RT and also a statistically significant effect on the interaction with reduction approaches. We noted a non trivial variability of achieved results (on average 89.5% for RF, 52.4% for RS, and 69.8% for RT). In particular, 2OPT, GRD and GR3D had the highest variability for RF (94.7% vs. an average of 85.6% of other approaches), while *MORE+* had the highest variability for RS (64% vs. an average of 50.5% of other approaches) and RT (78.7% vs. an average of 68.3% of other approaches).

In Table 18, we report the results of the Spearman’s correlation test obtained for such co-factors for *MORE+*, GR3D and for other approaches. We could observe that Reqs, NotTestedReqs, TCSRedundancy, TCSRevealing \geq 1fault and TCSRevealing1fault had a positive impact on RF for both *MORE+* and GR3D, even if the results were not statistically relevant in the case of GR3D. For the other approaches, Req, TCSRedundancy, TCSRevealing \geq 1fault had a negative impact on RF, while NotTestedReqs, TCSRedundancy still had a positive effect.

7 DISCUSSION

In this section, we first present overall considerations according to the investigated research questions. We conclude delineating possible implications of our research and its possible future extensions.

7.1 Overall Considerations

We observed that traditional test suite reduction approaches identified test suites smaller than those identified by applying evolutionary (multi-objective) approaches. This might be due to the fact that these approaches aim to better balance among different objectives. In other terms, even if all the considered traditional approaches consider at least code coverage and execution cost (while GR3D additionally considers also requirements coverage), it seems that code coverage remains the predominant objective that drive test suite reduction. Conversely, NSGA-II-based approaches aim to explicitly and equally balance among the considering objective, thus producing a Pareto front of candidate solutions. Among the traditional approaches, GR3D was the one that reduces less, this might be due to the fact that it tries to reduce according to three objectives instead of two, as well as done, for instance, by GRD. *MORE+* preserved the capability of reducing suites that characterize also traditional evolutionary and multi-objective approaches, i.e., the one that optimizes test suite reduction according to code coverage and execution cost. We observed that *MORE+* identified test suite reductions that cover less application code than other reduction approaches. However, giving more emphasis to the fault-prone parts of code and selecting suites that better balanced low- (coverage of code and execution cost) and high- level information (application requirement coverage), *MORE+* outperformed other approaches in terms of capability to detect faults, even if the achieved results was not statistically confirmed in the case of 2OPT. *MORE+* also outperformed a traditional NSGA-II-based approach that only considers low-level information (code coverage and execution time). Hence, we can delineate the following take-away outcomes: (i) automatic weighting of code and requirements applied when computing the coverage allows *MORE+* to find reduced suites that cover less application source code than traditional approaches, but these reduced suites better focus on fault-prone source code; (ii) the coverage of application (functional) requirements allows *MORE+* to find test suites that preserve the coverage of all the relevant business aspects of the application under testing; and (iii) the use of the NSGA-II-based algorithm tends to find reduced test suites that are more effective than those the greedy algorithm with three objectives (i.e., GR3D) finds.

On the basis of the observed results, we can state that test suites reduced by applying approaches that fill the gap between low- and high-level information (source code and requirements) preserve their capability in detecting faults as compared with corresponding unreduced test suites. Among these approaches, it seems that *MORE+* is generally more effective than others. Therefore, we can positively answer RQ1 stating that *MORE+* is effective in reducing test suites with respect to existing approaches.

Concerning the efficiency of the method (i.e., reduction cost and cost-effectiveness), experimental results suggested that the recovery of similarity links among software arti-

facts was costly and time consuming, so negatively affecting the overall suite reduction cost. On the other hand, we observed that *MORE+* was overall competitive with respect to the baseline approaches in terms of cost-effectiveness. Furthermore, an increase of the number of consecutive versions of the application under test indicated an increase in the competitiveness of *MORE+* in terms of cost-effectiveness. Summarizing, we cannot provide conclusive results on the efficiency of *MORE+*. Indeed, we could answer RQ2 stating that considering the suite reduction cost *MORE+* is not efficient with respect to baseline approaches, but *MORE+* is cost-effective with respect to them when we consider the reduction cost in relation to its fault detection capability.

Obtained results also highlighted some trends about how application artifacts might influence the capability of reduced test suites to detect faults. Even if not all these trends were supported by statistically significant results, we observed that if the number of test cases revealing at least one fault increases, reduction approaches tend to preserve the capability of the reduced test suites to detect faults. We also observed that an increase of functional redundancy decreases the capability of two-objectives traditional approaches to reduce test suites that preserve their capability to detect faults (as compared with the whole suites). Two trends differentiate results achieved by the traditional evolutionary and multi-objective approaches and *MORE+*. The first trend concerns the impact of fault density, while the second the size of test suites. Indeed, a trend seemed to exist and we can summarize it as follows: an increase of fault density (i.e., increase of the number of faults in few application requirements) or test suite size negatively influences the capability of traditional evolutionary approach in producing effective reduced test suites (i.e., that preserve the capability of finding faults), while they did not negatively impact on the results of our approach. We can postulate that the additional dimension (i.e., the functional) and the adoption of an automatic weighting scheme give *MORE+* the chance to better focus on relevant application parts (source code and application requirements).

7.2 Implications and Future Extensions

We focus on the researcher and the practitioner perspectives to discuss implications and possible future extensions for the research in the regression testing field.

- Test suite reduction might benefit from the use of evolutionary and multi-objective algorithms and automatic artifacts analysis and weighting approaches. That is, a test suite reduction obtained by applying *MORE+* is able to reveal more faults than a test suite reduced by applying baseline approaches. This result is relevant for practitioners interested in using *MORE+* in their company. The researcher could be interested in this result to better study the use of structural, functional, and cost dimensions in software regression testing. Definitely, the results presented in this paper and those

previously shown [3], [37] pose the basis for future research on this matter.

- Results indicated that our approach produces effective test suites, but it might be costly to apply if compared with the studied baselines. However, we observed a high cost-effectiveness of the suites reduced by *MORE+*, in fact, its cost-effectiveness is quite competitive even if it is applied to one version of the application to-be tested and that *MORE+* becomes even more competitive when it is applied on at least two subsequent versions of the same application to-be tested. This is clearly relevant for the practitioner interested in reducing the cost to perform regression testing and improving detect fault capability at the same time.
- This implication is related to the previous one since the practitioner has to take into account the execution cost to reduce test suites, when choosing a new approach. In our case, there is a cost due to the application of LSI to compute similarity links among software artifacts. Although this cost seems to be adequately paid back (see RQ2), the researcher could be interested in studying different text retrieval models and techniques to reduce the time needed to compute similarities among software artifacts.
- Different kinds of applications were considered in our study. These applications were realistic enough for small- to medium-sized software projects and faults where differently distributed within code and requirements. Although we are not sure that the achieved results scale to real commercial projects, obtained results seem to reassure us about their generalizability. Indeed, the magnitude of the benefits deriving from the use of *MORE+* suggests that our outcomes could be generalized also to applications from different application domains. These points surely deserve further investigations and they are relevant from both the practitioner and the researcher perspectives.
- More traditional reduction approaches are superior to multi-objective approaches to identify smaller test suites, while they are inferior with respect to their capability in revealing faults. This result is clearly relevant also for the practitioner, who has to choose the most suitable approach to be used in his/her company.
- An empirical assessment of a new technology/method makes easier and faster its transfer to software industry [79]. This might happen when empirical results show that such a technology/method effectively solves actual issues. This is the case of our proposed solution. We can also postulate that the use of *MORE+* does not require a radical process change in a given software company (e.g., similarity links are automatically recovered and covered information is easy to obtain by instrumenting source code). This point is of interest primarily for the practitioner. On the other hand, the researcher could be interested to conduct users' studies to assess how actual testers perceive *MORE+* and if they might have some benefits from its use. This kind

of investigation makes sense because of our results and represents a future direction for our research.

8 CONCLUSION

In this paper, we proposed and studied *MORE+*, a Multi-Objective test cases REduction approach. It is a multi-objective approach that reduces test suites by considering coverage of both source code and application requirements. The cost to execute test cases is also taken into account. An IR-based textual similarity approach was applied to link different kinds of application artifacts (i.e., requirements specifications, source code, and test cases). A reduced test suite is then determined by using a multi-objective optimization, implemented in terms of NSGA-II. *MORE+* has been evaluated through a large empirical assessment on 20 open-source Java applications. Results proved the effectiveness of *MORE+* in reducing test suites with respect to seven baseline approaches well known in the regression testing field. Results also suggested that *MORE+* is not efficient in reducing test suites as compared with the studied baseline approaches but it is cost-effective with respect to these approaches. This is especially true when reduced test suites are used to test at least two subsequent versions of the same application under test.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their helpful and constructive comments that greatly contributed to improve the final version of the manuscript. They would also like to thank the Editors for their generous comments and support during the entire review process.

REFERENCES

- [1] J. Karlsson and K. Ryan, "A cost-value approach for prioritizing requirements," *IEEE Software*, vol. 14, pp. 67–74, 1997.
- [2] 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, 2010.
- [3] A. Marchetto, M. M. Islam, W. Asghar, A. Susi, and G. Scanniello, "A multi-objective technique to prioritize test cases," *IEEE Transaction on Software Engineering*, vol. 42, no. 10, pp. 918–940, October 2016.
- [4] M. Gligoric, L. Eloussi, and D. Marinov, "Practical regression test selection with dynamic file dependencies," in *Proc. of International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: ACM, 2015, pp. 211–222.
- [5] A. Gonzalez-Sanchez, R. Abreu, H. G. Gross, and A. J. C. van Gemund, "Prioritizing tests for fault localization through ambiguity group reduction," in *Proc. of International Conference on Automated Software Engineering*, 2011, pp. 83–92.
- [6] G. Rothermel, S. Elbaum, A. Malishevsky, and P. Kallakuri, "On test suite composition and cost-effective regression testing," *ACM Transactions on Software Engineering and Methodology*, vol. 13, pp. 277–331, 2004.
- [7] D. Jeffrey and N. Gupta, "Improving fault detection capability by selectively retaining test cases during test suite reduction," *IEEE Transaction on Software Engineering*, vol. 33, no. 2, pp. 108–123, Feb. 2007.
- [8] A. M. Smith, J. Geiger, G. M. Kapfhammer, and M. L. Soffa, "Test suite reduction and prioritization with call trees," in *Proc. International Conference on Automated Software Engineering*. ACM, 2007, pp. 539–540.
- [9] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid, "An empirical study of junit test-suite reduction," in *Proc. of International Symposium on Software Reliability Engineering*. IEEE Computer Society, Nov 2011, pp. 170–179.
- [10] H. Zhong, L. Zhang, and H. Mei, "An experimental comparison of four test suite reduction techniques," in *Proc. of International Conference on Requirements Engineering*. New York, USA: ACM, 2006, pp. 636–640.
- [11] J. Campos and R. Abreu, "Leveraging a constraint solver for minimizing test suites," in *Proc. of International Conference on Quality Software*, July 2013, pp. 253–259.
- [12] S. McMaster and A. M. Memon, "Call stack coverage for test suite reduction," in *Proc. of International Conference on Software Maintenance*. IEEE Computer Society, 2005, pp. 539–548.
- [13] G. Rothermel, M. J. Harrold, J. von Ronne, and C. Hong, "Empirical studies of test-suite reduction," *Software Testing, Verification and Reliability*, vol. 12, no. 4, pp. 219–249, 2002.
- [14] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, "Effect of test set minimization on fault detection effectiveness," in *Proc. of International Conference on Software Engineering*. ACM, april 1995, p. 41.
- [15] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proc. of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 435–445.
- [16] S. Yoo and M. Harman, "Pareto efficient multi-objective test case selection," in *Proc. of International Symposium on Software testing and Analysis*. ACM, 2007, pp. 140–150.
- [17] A. Shi, A. Gyori, M. Gligoric, A. Zaytsev, and D. Marinov, "Balancing trade-offs in test-suite reduction," in *Proc. of International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 246–256.
- [18] X. MA, Z. He, B. kui Sheng, and C. Ye, "A genetic algorithm for test-suite reduction," in *Proc. of International Conference on Systems Man and Cybernetics*, vol. 1, Oct. 2005, pp. 133–139 Vol.1.
- [19] L. de Souza, P. de Miranda, R. Prudencio, and F. de Barros, "A multi-objective particle swarm optimization for test case selection based on functional requirements coverage and execution effort," in *Proc. of Conference on Tools with Artificial Intelligence*. IEEE Computer Society, nov. 2011, pp. 245–252.
- [20] A. Panichella, R. Oliveto, M. D. Penta, and A. D. Lucia, "Improving multi-objective test case selection by injecting diversity in genetic algorithms," *IEEE Transaction on Software Engineering*, vol. 41, no. 4, pp. 358–383, 2015.
- [21] A. Marchetto, M. M. Islam, A. Susi, and G. Scanniello, "A multi-objective technique for test suite reduction," in *Proc. of International Conference on Software Engineering Advances*, 2013, pp. 18–24.
- [22] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.
- [23] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman, "Indexing by Latent Semantic Analysis," *Journal of the American Society of Information Science*, vol. 41, no. 6, pp. 391–407, 1990.
- [24] T. Chen and M. Lau, "Dividing strategies for the optimization of a test suite," *Information Processing Letters*, vol. 60, no. 3, pp. 135–141, 1996.
- [25] A. Malishevsky, J. Ruthruff, G. Rothermel, and S. Elbaum, "Cost-cognizant Test Case Prioritization," University of Nebraska, Department of Computer Science and Engineering, Tech. Rep., March 2006.
- [26] A. M. Smith and G. M. Kapfhammer, "An empirical study of incorporating cost into test suite reduction and prioritization," in *Proc. of Annual ACM Symposium on Applied Computing*. ACM, 2009.
- [27] C.-T. Lin, K.-W. Tang, and G. M. Kapfhammer, "Test suite reduction methods that decrease regression testing costs by identifying irreplaceable tests," *Information and Software Technology*, vol. 56, no. 10, pp. 1322 – 1344, 2014.
- [28] S. Selvakumar and N. Ramaraj, "Regression test suite minimization using dynamic interaction patterns with improved FDE," *European Journal of Scientific Research*, vol. 49, no. 3, pp. 332–353, 2011.
- [29] A. Gotlieb and D. Marijan, "FLOWER: optimal test suite reduction as a network maximum flow," in *Proc. of International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 171–180.
- [30] A. Shi, T. Yung, A. Gyori, and D. Marinov, "Comparing and combining test-suite reduction and regression test selection," in *Proc.*

- of the Joint Meeting on Foundations of Software Engineering, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 237–247.
- [31] A. Groce, M. A. Alipour, C. Zhang, Y. Chen, and J. Regehr, “Cause reduction for quick testing,” in *Proceedings of IEEE International Conference on Software Testing, Verification, and Validation*, ser. ICST ’14. Washington, DC, USA: IEEE, 2014, pp. 243–252.
- [32] M. A. Alipour, A. Shi, R. Gopinath, D. Marinov, and A. Groce, “Evaluating non-adequate test-case reduction,” in *Proc. of IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: ACM, 2016, pp. 16–26.
- [33] S. Mirarab, S. Akhlaghi, and L. Tahvildari, “Size-constrained regression test case selection using multicriteria optimization,” *IEEE Transactions on Software Engineering*, vol. 38, no. 4, pp. 936–956, July 2012.
- [34] T. Back, *Evolutionary Algorithm: Theory and Practice*. Oxford University Press, 1996.
- [35] D. Fogel, *Evolutionary Computation*. IEEE Press, 1995.
- [36] J. Costa, “A Multi-Objective Approach to Test Suite Reduction,” Master’s thesis, Faculdade de Engenharia da Universidade do Porto, Portugal, 2015.
- [37] M. M. Islam, A. Marchetto, A. Susi, and G. Scanniello, “A multi-objective technique to prioritize test cases based on latent semantic indexing,” in *Proc. of European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, March 2012, pp. 21–30.
- [38] O. C. Z. Gotel and C. W. Finkelstein, “An Analysis of the Requirements Traceability Problem,” in *Proc. of International Conference on Requirements Engineering*, 1994, pp. 94–101.
- [39] S. Klock, M. Gethers, B. Dit, and D. Poshyanyk, “Traceclipse: an eclipse plug-in for traceability link recovery and management,” in *Proc. of International Workshop on Traceability in Emerging Forms of Software Engineering*. ACM, 2011, pp. 24–30.
- [40] M. Lormans and A. van Deursen, “Reconstructing requirements coverage views from design and test using traceability recovery via lsi,” in *Proc. of International Workshop on Traceability in Emerging Forms of Software Engineering*. ACM, 2005, pp. 37–42.
- [41] A. Qusef, R. Oliveto, and A. De Lucia, “Recovering traceability links between unit tests and classes under test: An improved method,” in *Proc. of International Conference on Software Maintenance*. IEEE Computer Society, 2010, pp. 1–10.
- [42] X. Zou, R. Settini, and J. Cleland-Huang, “Improving automated requirements trace retrieval: a study of term-based enhancement methods,” *Empirical Software Engineering*, vol. 15, pp. 119–146, April 2010.
- [43] A. Marcus and J. I. Maletic, “Recovering documentation-to-source-code traceability links using latent semantic indexing,” in *Proc. of International Conference on Software Engineering*. IEEE Computer Society, 2003, pp. 125–137.
- [44] S. K. Sundaram, J. H. Hayes, A. Dekhtyar, and E. A. Holbrook, “Assessing traceability of software engineering artifacts,” *Requirements Engineering*, vol. 15, no. 3, pp. 313–335, 2010.
- [45] C. D. Manning, P. Raghavan, and H. Schütze, *An Introduction to Information Retrieval*. Cambridge University Press, England, 2009.
- [46] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, “Bug localization using latent dirichlet allocation,” *Information & Software Technology*, vol. 52, no. 9, pp. 972–990, Sep. 2010.
- [47] A. Abadi, M. Nisenson, and Y. Simionovici, “A traceability technique for specifications,” in *Proc. of International Conference on Program Comprehension*. IEEE CS Press, 2008, pp. 103–112.
- [48] S. Wang, D. Lo, Z. Xing, and L. Jiang, “Concern localization using information retrieval: An empirical study on linux kernel,” in *Proc. of Working Conference on Reverse Engineering*. IEEE CS Press, 2011, pp. 92–96.
- [49] G. Booch, J. Rumbaugh, and I. Jacobson, *Unified Modeling Language User Guide, The (2Nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005.
- [50] A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora, “Recovering traceability links in software artifact management systems using information retrieval methods,” *ACM Transactions on Software Engineering and Methodology*, vol. 16, no. 4, 2007.
- [51] Y. Zhang and M. Harman, “Search Based Optimization of Requirements Interaction Management,” in *Proc. of International Symposium on Search Based Software Engineering*. IEEE Computer Society, 2010, pp. 47–56.
- [52] M. Asghar, A. Marchetto, A. Susi, and G. Scanniello, “Maintainability-Based Requirements Prioritization by Using Artifacts Traceability and Code Metrics,” in *Proc. of European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, 2013, pp. 417–420.
- [53] R. Lincke, J. Lundberg, and W. Löwe, “Comparing Software Metrics Tools,” in *Proc. of Symposium on Software Testing and Analysis*. ACM, 2008, pp. 131–142.
- [54] S. R. Chidamber and C. F. Kemerer, “A Metrics Suite for Object Oriented Design,” *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, Jun. 1994.
- [55] M. Eaddy, T. Zimmermann, K. D. Sherwood, V. Garg, G. C. Murphy, N. Nagappan, and A. V. Aho, “Do Crosscutting Concerns Cause Defects?” *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 497–515, Jul. 2008.
- [56] B. C. da Silva, C. Sant’Anna, and C. Chavez, “Concern-based cohesion as change proneness indicator: an initial empirical study,” in *Proc. of Workshop on Emerging Trends in Software Metrics*. ACM, 2011, pp. 52–58.
- [57] V. K. Mathur, “How well do we know pareto optimality?” *The Journal of Economic Education*, vol. 22, no. 2, pp. 172–178, 1991.
- [58] E. Zitzler, M. Laumanns, and L. Thiele, “SPEA 2: Improving the Strength Pareto Evolutionary algorithm,” in *Technical report 103.*, 2001, pp. 1–21.
- [59] M. Clerc and J. Kennedy, “The particle swarm - explosion, stability, and convergence in a multidimensional complex space,” *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 1, pp. 58–73, Sep. 2002.
- [60] S. Kirkpatrick, C. Gelatt, and M. Vecchi, “Optimization by simulated annealing,” *Science*, vol. 220, no. 4598, pp. 671–680, May 1983.
- [61] A. Marchetto, C. Di Francescomarino, and P. Tonella, “Optimizing the trade-off between complexity and conformance in process reduction,” in *Proc. of International Conference on Search Based Software Engineering*. Springer-Verlag, 2011, pp. 158–172.
- [62] P. Tonella, A. Marchetto, C. D. Nguyen, Y. Jia, K. Lakhotia, and M. Harman, “Finding the optimal balance between over and under approximation of models inferred from execution logs,” in *Proc. of International Conference on Software Testing, Verification and Validation*. IEEE Computer Society, 2012.
- [63] J. J. Durillo and A. J. Nebro, “jMetal: A Java Framework for Multi-Objective Optimization,” *Advances in Engineering Software*, vol. 42, no. 10, pp. 760 – 771, 2011.
- [64] V. R. Basili and R. W. Selby, “Comparing the effectiveness of software testing strategies,” *IEEE Transactions on Software Engineering*, vol. 13, pp. 1278–1296, December 1987.
- [65] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov, “Comparing non-adequate test suites using coverage criteria,” in *Proc. of the International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM, 2013, pp. 302–313.
- [66] R. Gopinath, C. Jensen, and A. Groce, “Code coverage for suite evaluation by developers,” in *Proc. of the International Conference on Software Engineering*. New York, NY, USA: ACM, 2014, pp. 72–82.
- [67] H. Leung and L. White, “Insights into regression testing,” in *Proc. of International Conference on Software Maintenance*. IEEE Computer Society, 1989, pp. 60–69.
- [68] A. Malishevsky, G. Rothermel, and S. Elbaum, “Modeling the cost-benefits tradeoffs for regression testing techniques,” in *Proc. of International Conference on Software Maintenance*. IEEE Computer Society, 2002, pp. 204–213.
- [69] A. Marchetto, F. Ricca, and P. Tonella, “An empirical validation of a web fault taxonomy and its usage for web testing,” *Journal of Web Engineering*, vol. 8, no. 4, pp. 316–345, 2009.
- [70] A. Marchetto and P. Tonella, “Using search-based algorithms for ajax event sequence generation during testing,” *Empirical Software Engineering*, vol. 16, no. 1, pp. 103–140, 2011.
- [71] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Springer, 2012.
- [72] A. Arcuri, “It really does matter how you normalize the branch distance in search-based software testing,” *Journal of Verification and Reliability*, vol. 23, pp. 119–147, 2013.
- [73] R. Baker, “Modern permutation test software,” in *E. Edgington Randomization Tests*, New York, Marcel Decker, 1995.
- [74] F. J. Anscombe, “Sequential estimation,” *Journal of the Royal Statistical Society. Series B (Methodological)*, pp. 1–29, 1953.
- [75] C. Spearman, “The proof and measurement of association between two things,” *American Journal of Psychology*, vol. 15, pp. 88–103, 1904.

- [76] Y. Benjamini and Y. Hochberg, "Controlling the false discovery rate: A practical and powerful approach to multiple testing," *Journal of the Royal Statistical Society Series B (Methodological)*, vol. 57, no. 1, pp. 289–300, 1995.
- [77] E. Murphy-Hill, T. Zimmermann, and N. Nagappan, "Cowboys, ankle sprains, and keepers of quality: How is video game development different from software development?" in *Proc. of International Conference on Software Engineering*. New York, NY, USA: ACM, 2014, pp. 1–11.
- [78] K. Deb, *Multi-Objective Optimization using Evolutionary Algorithms*. UK: Wiley Chichester, 2001.
- [79] S. L. Pfleeger and W. Menezes, "Marketing technology to software practitioners," *IEEE Software*, vol. 17, no. 1, pp. 27–33, 2000.



Angelo Susi is a research scientist in the Software Engineering group at Fondazione Bruno Kessler in Trento, Italy. His research interests are in the areas of requirements engineering, goal-oriented software engineering, formal methods for requirements validation, and search-based software engineering. He published more than 100 refereed papers in journals and international conferences such as TSE, TOSEM, IST, SoSyM, FSE, ICSE, RE. He participated in the organization committee of several conferences, such as SSBSE'12 (General Chair), RE'11 (Local and Financial chair) and in program committees of international conferences and workshops (such as ICSE, RE, REFSQ, CAiSE and SSBSE). He also served as reviewer for several Journals such as TSE, REJ, IST, JSS. He has been the scientific manager of the EU FP7 project RISCOSS.



Alessandro Marchetto is currently an independent researcher working in the field of Software Engineering. He received his PhD degree in Software Engineering from the University of Milano, Italy in 2007. From 2006 till the end of 2012 he was a researcher at the Center for Information Technology (CIT) of the Bruno Kessler Foundation in Trento, Italy, working with the Software Engineering group. His primary research interests concern Software Engineering and, in particular,

include quality, verification and testing of Software Systems and of Internet-based systems. He published more than 85 papers in primary international conferences and journals. He regularly reviews papers for international journals (e.g., TSE, IST, JSS, EMSE, IET) and conferences (e.g., ICME, CSMR/SANER, ICPC). He collaborated to the organization of more than 10 international scientific events (e.g., SSBSE 2012, SCAM 2012, EmpiRE 2011-2012-2013-2014, WSE 2008-2012).



Giuseppe Scanniello received his Laurea and Ph.D. degrees, both in Computer Science, from the University of Salerno, Italy, in 2001 and 2003, respectively. In 2006, he joined, as an Assistant Professor, the Department of Mathematics and Computer Science at the University of Basilicata, Potenza, Italy. In 2015, he became an Associate Professor at the same university. His research interests include requirements engineering, empirical software engineering, reverse engineering, reengineering, software visualization, workflow automation,

migration, wrapping, integration, testing, green software engineering, global software engineering, cooperative supports for software engineering, visual languages and e-learning. He has published more than 150 referred papers in journals, books, and conference proceedings. He serves on the organizing of major international conferences (as general chair, program co-chair, proceedings chair, and member of the program committee) and workshops in the field of software engineering (e.g., ICSE, ASE, ICSME, ICPC, SANER, and many others). Giuseppe Scanniello leads both the group and the laboratory of software engineering at the University of Basilicata. He recently obtained the Italian National Scientific Qualification as Full Professor in Computer Science. He is a member of IEEE and IEEE Computer Society. More on <http://www2.unibas.it/gscanniello/>