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 differences 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 version [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
approaches in preserving a high capability to discover faults. In particular, it is able to find in between 5.9 and 13.4 percent (with an average of 10.2 percent) 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 regression 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 percent 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 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

---

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

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]). Fig. 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-known and used techniques we can find: Latent Semantic Indexing [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 preliminary 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 \( C C^T \) 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 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 $S_{\text{red}}$, 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, $S_{\text{red}}$ 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 Fig. 2. This process is composed of the following main steps:

1) Recovering Similarity Links. MORE+ recovers textual similarity links (i.e., $\text{SimilarityLinks}$ in Fig. 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 ($\text{SimilarityLinks}$) 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 ($\text{Weights}$) 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—$\text{CoveredStatements}$—and execution time—$\text{ExecutionTime}$) and similarity links recovery ($\text{SimilarityLinks}$). In this phase, the test cases in the original test suite $TestCases$ are also needed to identify $\text{ReducedTestSuite}$.

In the following sections, 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 ($\text{SimilarityLinks}$) 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., \( : \text{Reqs} : \text{Code} \), respectively). We adopted an approach inspired by the one of Lincke et al. [53] to measure a Maintainability Index (\( MI_{\text{class}} \)) 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-proveness of each class, thus for prioritizing classes. Similarly, MORE+ computes a Maintainability Index (\( MI_{\text{req}} \)) for each requirement and it uses this index to prioritize requirements. That is, this index is used to estimate more fault-proveness requirements. These orderings are exploited to select test cases in the test suite that exercise the most critical (fault-proveness) classes and requirements, i.e., top-ranked for \( MI_{\text{class}} \) and \( MI_{\text{req}} \).

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 sections.

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 property, the reference to the software property it measures, and its definition.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Ref.</th>
<th>Property</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(CBO) Coupling Between Objects</td>
<td>[54]</td>
<td>Coupling</td>
<td>It is the number of classes to which a class is coupled</td>
</tr>
<tr>
<td>(RFC) Response For a Class</td>
<td>[54]</td>
<td>Coupling</td>
<td>It is the set of methods that can potentially be executed in response to a message received by an object of the class</td>
</tr>
<tr>
<td>(LOCM) Lack Of Cohesion on Methods</td>
<td>[54]</td>
<td>Cohesion</td>
<td>It describes the lack of cohesion among methods of a class</td>
</tr>
<tr>
<td>(LOCs) Lines Of Code</td>
<td>-</td>
<td>Size</td>
<td>It counts the lines of code of a class</td>
</tr>
<tr>
<td>(CBO) Number of methods</td>
<td>-</td>
<td>Size</td>
<td>It counts the number of methods of a class</td>
</tr>
<tr>
<td>(DIT) Depth of Inheritance Tree</td>
<td>[54]</td>
<td>Inheritance</td>
<td>It is the length of the class from the root of the inheritance tree</td>
</tr>
<tr>
<td>(NOC) Number of Children</td>
<td>[54]</td>
<td>Complexity</td>
<td>It is the number of immediate subclasses of the class in the class hierarchy</td>
</tr>
<tr>
<td>(MCC) McCabe Cyclomatic Complexity</td>
<td>[54]</td>
<td>Complexity</td>
<td>It is (median of) the number of flows thought the code of the method of a class</td>
</tr>
<tr>
<td>(WMC) Weighted Methods per Class</td>
<td>[54]</td>
<td>Complexity</td>
<td>It is the sum of the MCC for all methods in a class</td>
</tr>
</tbody>
</table>

To measure \( MI_{\text{class}} \) for each class (in : \text{Code} of the application under test, we adopt the object-oriented class-level metrics (see Table 1 top) adopted in [53]. Instead, to measure \( MI_{\text{req}} \) for each application requirement (\( : \text{Reqs} \)), 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 percent 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 \( \text{CBO}(c1) = 52 \) and \( \text{CBO}(c2) = 35 \), then \( c1 \) is an outlier for CBO (i.e., the value of \( c1 \) is in the range 85–100 percent 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...
TABLE 2
Software Quality Model (the Metrics and Weights W)

<table>
<thead>
<tr>
<th>Metric</th>
<th>Class-level Model</th>
<th>Requirements-level Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBO</td>
<td>2</td>
<td>NC</td>
</tr>
<tr>
<td>RFC</td>
<td>2</td>
<td>CDC+</td>
</tr>
<tr>
<td>LCOM</td>
<td>1</td>
<td>ShR</td>
</tr>
<tr>
<td>LOCs</td>
<td>1</td>
<td>ShR+</td>
</tr>
<tr>
<td>NOM</td>
<td>2</td>
<td>IN</td>
</tr>
<tr>
<td>DIT</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>NOC</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>MCC</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>WMC</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>W</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Maintainability index computation

MI of all for each and Maintainability index for requirements is the

\[ MI = \frac{\sum_{\mu \in \text{Metrics}} (W_\mu \ast O_\mu)}{\sum_{\mu \in \text{Metrics}} (W_\mu)} \]

Metrics is the set of metrics in the models, \( W_\mu \) is the
weight of the metric \( \mu \) and \( O_\mu \) is 0 if element is not an
outlier for the metric \( \mu \), while it is 1 if element is an out-
lier for \( \mu \). For instance, if a class \( cl \) is an outliers only
for CBO, LOCs, and DIT then \( MI(cl) = \frac{5}{13} = 0.385 \).
Therefore, \( cl \) has 0.385 (i.e., 38.5 percent) as index.

1) Maintainability index for classes. To compute the
maintainability index \( MI_{class}(c) \), we use only
class-level metrics.

2) Maintainability index for requirements. To compute
the maintainability index \( MI_{req}(r) \) for each require-
ment \( r \), two (sub-) indexes have to be computed,
\( MI_C(r) \) and \( MI_R(r) \). The former is computed by
averaging the index \( MI_{class} \) of all classes implement-
ing \( r \), while the latter is com-
puted by applying requirements-level metrics to
the classes implementing \( r \). In both the cases,
similarity links (\( \sim \) 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 imple-
menting \( r \) 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). Accord-
ing to this prioritization, we give a different weight to each
code class and requirement of the application when com-
puting their coverage for a test case. In detail, the weight we
use to prioritize the coverage of application code (see below \( w_r \) in Eq. (2)) and the weight we use to prioritize the cover-
age of application requirements (see below \( w_r \) in Eq. (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 (\( \text{ReducedTestSuite} \)) on several dimensions could be
exensive 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 dimen-
sions of the problem) have to be optimized simultaneously
to achieve reasonably good solutions. In this kind of prob-
lem, there is not a unique solution that optimizes each con-
sidered 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 solu-
tions is generated from an evolved population. In terms of
complexity, the time complexity of the NSGA-II algorithm
is \( O(np^2) \), where \( n \) 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 solu-
tions, 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 possi-
table 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 itera-
tion. The suites in the population are evaluated with
respect to each of the three objectives (code and
requirements coverage and execution time) and the fit-
ness 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 the 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 \( S_{\text{red}} \) of the whole test suite \( S \). The solution space for test suite reduction is given by all \( S_{\text{red}} \). For instance, given a test suite \( S \) composed of the test cases \( t_1, t_2, \) and \( t_3 \), possible reduced suites \( S_{\text{red}} \) are: \( \{t_1, t_2\}, \{t_1, t_3\}, \{t_1\}, \{t_2\}, \) and \( \{t_3\} \), 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 percent 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 \( \text{cumCov}(\text{red}_S) \)), the overall application requirements coverage of a reduced suite \( S \) (namely \( \text{cumKcov}(\text{red}_S) \)), and the overall tests case execution cost of a reduced suite \( S \) (namely \( \text{cumCost}(\text{red}_S) \)).

In the next sections, 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 (:\( \text{CoveredStatements} \)) 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 (:\( \text{TestCases} \)) and define the Weighted Code Coverage measure \( \text{WCCov}(t) \), for a given test case, as follows:

\[
\text{WCCov}(t) = \sum_{s \in \text{Statements}} \left\{ \begin{array}{ll}
w_c & s \in \text{CodeCovered} \\
0 & \text{otherwise}
\end{array} \right.
\]  

where the set \( \text{Statements} \) contains code statements (:\( \text{Code} \)); \( \text{CodeCovered} \) 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 (:\( \text{Weights} \)) by estimating the maintainability of each class. This allowed us to prioritize classes according to their maintainability when computing coverage. In other terms,
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

\[
\text{cumRCov}(S) = \sum_{t \in S} WRCov(t).
\]

- 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:

\[
RelativeCost(t) = Cost(S) - Cost(t).
\]

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

\[
\text{cumCost}(S) = \sum_{t \in S} RelativeCost(t).
\]

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, delayed greedy, traditional greedy, and 2-optimal greedy [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

\[
\text{ratio}(t) = \frac{\text{codeCoverage}(t)}{\text{executionCost}(t)},
\]

where \( t \) is the test case under evaluation, \( \text{codeCoverage}(t) \) is the code covered by \( t \) when and \( \text{executionCost}(t) \) 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 NSGAIIwp). It is a standard NSGA-II approach that considers: code coverage and execution cost. The implementation of the
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 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:

\[
RS(S_{red}) = \frac{|S| - |S_{red}|}{|S|} \times 100. \tag{12}
\]

$RS(S_{red})$ 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:

\[
RF(S_{red}) = \frac{|F| - |F_{red}|}{|F|} \times 100, \tag{13}
\]

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:

\[
RT(S_{red}) = \frac{time(S) - time(S_{red})}{time(S)} \times 100, \tag{14}
\]

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

### Table 3: Objects Under Study

<table>
<thead>
<tr>
<th>App.</th>
<th>Size (LOCs)</th>
<th>Test Cases</th>
<th>Req.</th>
<th>Faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>LaTazza</td>
<td>2 k</td>
<td>33</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>AveCalc</td>
<td>2 k</td>
<td>47</td>
<td>10</td>
<td>15</td>
</tr>
<tr>
<td>CommonsProxy</td>
<td>5 k</td>
<td>179</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>DBUtis</td>
<td>5 k</td>
<td>225</td>
<td>12</td>
<td>14</td>
</tr>
<tr>
<td>iTrust</td>
<td>15 k</td>
<td>919</td>
<td>15</td>
<td>21</td>
</tr>
<tr>
<td>CommonsCodec</td>
<td>17 k</td>
<td>608</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td>JTeddy</td>
<td>20 k</td>
<td>289</td>
<td>25</td>
<td>15</td>
</tr>
<tr>
<td>Woden</td>
<td>22 k</td>
<td>263</td>
<td>24</td>
<td>19</td>
</tr>
<tr>
<td>Log4j</td>
<td>25 k</td>
<td>1,029</td>
<td>24</td>
<td>20</td>
</tr>
<tr>
<td>JXPath</td>
<td>25 k</td>
<td>386</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>CommonsIO</td>
<td>25 k</td>
<td>859</td>
<td>18</td>
<td>20</td>
</tr>
<tr>
<td>CommonsBcel</td>
<td>30 k</td>
<td>75</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>CommonsBeanUtils</td>
<td>32 k</td>
<td>1,556</td>
<td>26</td>
<td>22</td>
</tr>
<tr>
<td>xmGraphics</td>
<td>34 k</td>
<td>196</td>
<td>24</td>
<td>15</td>
</tr>
<tr>
<td>xSecurity</td>
<td>40 k</td>
<td>92</td>
<td>23</td>
<td>15</td>
</tr>
<tr>
<td>CommonsCollections</td>
<td>50 k</td>
<td>798</td>
<td>17</td>
<td>20</td>
</tr>
<tr>
<td>Pmd</td>
<td>55 k</td>
<td>698</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>CommonsLang</td>
<td>60 k</td>
<td>2,307</td>
<td>16</td>
<td>20</td>
</tr>
<tr>
<td>Jabref</td>
<td>70 k</td>
<td>213</td>
<td>31</td>
<td>20</td>
</tr>
<tr>
<td>Xerces</td>
<td>138 k</td>
<td>376</td>
<td>20</td>
<td>20</td>
</tr>
</tbody>
</table>
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 \( \text{Div} \). 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 \( \text{Div}(S_1, S_2) \) means that suites have a high number of test cases in common. The higher the \( \text{Div}(S_1, S_2) \) 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 \( S: \text{Code Cov}(S) \) and \( \text{Reqs Cov}(S) \). 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 \( \text{Code Cov} \) value, the higher the code coverage is. Similarly, the higher the \( \text{Reqs Cov} \), 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 \( \text{Cost}_{\text{Red}} \), 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 (\( \text{CostEff} \)). 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 \( S_{\text{red}} \) (used to test \( g \) versions of an application), we can define:

\[ \begin{align*}
(\text{i}) \quad \text{Ca}(S) & \quad \text{as the cost of the analysis conducted to prepare test suite execution and reduction (e.g., textual-based similarity recovery links);} \\
(\text{ii}) \quad \text{Ce}(S) & \quad \text{as the execution cost of test cases}; \\
(\text{iii}) \quad \text{Cc}(S) & \quad \text{as the cost of result checking}; \\
(\text{iv}) \quad \text{Cr}(S) & \quad \text{as the cost for the execution of a test suite reduction approach (i.e., effort/time required to obtain the reduced suite);} \\
(\text{v}) \quad \text{Nm}(S) & \quad \text{as the maintenance cost of a test suite (e.g., effort and time for fixing a suite);} \\
(\text{vi}) \quad \text{Cf}(S_{\text{red}}) & \quad \text{as the cost of omitting faults by not selecting the set \( S \setminus S_{\text{red}} \) where \( F_k \) is the set of faults detected in the release \( k \).}
\end{align*} \]

\[ C = g * \text{Ce}(S) + g * \text{Ce}(S) + g * \text{Nm}(S). \]  \hspace{1cm} (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

\[ \text{CostEff} = \text{Ca}(S) + g * \text{Ce}(S_{\text{red}}) + g * \text{Cc}(S_{\text{red}}) + \text{Cr}(S) + \sum_{1 \leq k \leq g} \left( \text{Cf}(F_k(S) \setminus F_k(S_{\text{red}})) \right) + g * \text{Nm}(S_{\text{red}}). \]  \hspace{1cm} (16)

We excluded from our analysis the cost of: (i) results checking \( \text{Ca}(S) \) and (ii) suite maintenance \( \text{Nm}(S_{\text{red}}) \). This was needed because this information was not available. Furthermore, we computed the cost of omitting faults \( \text{Ff}(F_k(S) \setminus F_k(S_{\text{red}})) = fc * |F_k| \), where \( F_k \) 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 \( \text{NSGAII}_{2a} \) 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 \( \text{NSGAII}_{2a} \) 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 \( \text{Faults} \) 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 failure to analyze the code of the application and to get information about how to restore that fault. For instance, Fig. 3a shows an example of a fault description posted in the DbUtil bug-tracker 2.

Executing test suites and collecting data

We tested the following hypothesis:

\[ NH_{A} - \text{there is no difference in values of } m \text{ (e.g., } RS, RF, RT \text{ and } F) \text{ computed on test suites reduced by applying } MORE+ \text{ and baselines.} \]

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 \([72]\). This statistical 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).

Analyzing data. We analyzed collected data. We performed statistical analyses when needed. In the case of the evolutionary approaches (i.e., \( MORE+ \) and \( NSGAI_{I}^{(i)} \)), 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:

\[ NH_{A} - \text{there is no difference in values of } m \text{ (e.g., } RS, RF, RT \text{ and } F) \text{ computed on test suites reduced by applying } MORE+ \text{ and baselines.} \]
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 1,000 or 5,000) and (ii) stopping when the estimated standard deviation of the p-value is under a given amount, often 10 percent, 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 5,000 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 percent of committing a Type-I-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 sections, 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 $A_{12}$ 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 percent (see Table 4) and the (average) RS medians for other approaches (HGS, DGR, GRD, and 2OPT) are 79.7 percent. This result is also confirmed by the $A_{12}$ 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 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 based on evolutionary algorithms (MORE+ and NSGAII$_{2d}$) is 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$_{2d}$.
returned 0.001 as the p-value), while the results of both the Mann-Whitney and the $A_{d2}$ ($>0.57$) tests suggest that MORE+ achieved better results than other approaches (NSGAII 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 $A_{d2}$ 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 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 percent, for 2OPT is 52.5 percent, while the average of median values for HGS, DGR, GRD NSGAII$_{2d}$ and GR3D is 56.7 percent. 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 NSGAII$_{2d}$ 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 GRD 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+ and NSGAII$_{2d}$, and MORE+ and 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 percent for our approach, while the (average) median value for these approaches is 73.9 percent. Results are confirmed by the $A_{d2}$ 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 percent 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 percent of test cases in contrast to 36.4 percent shared between MORE+ and NSGAII$_{2d}$ and 29.5 percent (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 Table 7 and Table 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 \( \text{NSGAII}_{2d} \). For instance, for \( \text{MORE}+ \), the mean values of Code_Cov was 88 percent and Reqs_Cov was 96.8 percent (see Table 7), for \( \text{NSGAII}_{2d} \), it was respectively 88.9 percent and 95.1 percent, while the (average) mean values of Code_Cov and Reqs_Cov for other traditional approaches were 98.7 percent and 90.6 percent respectively.

As a further analysis, for each application object of the study, we compared both code and requirement coverage of test suites reduced by \( \text{MORE}+ \), \( \text{NSGAII}_{2d} \), 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 percent), suites reduced by \( \text{MORE}+ \) covered less code than traditional approaches, but in 12 of such applications (70.5 percent) \( \text{MORE}+ \) discovered more faults than traditional approaches. Similarly, we observed that on 8 out of 20 applications (40 percent) \( \text{MORE}+ \) covered more requirements than traditional approaches and that in 7 of such applications (35 percent) it allowed us to discover more faults. With respect to suites reduced by \( \text{NSGAII}_{2d} \), the suites reduced by \( \text{MORE}+ \) covered slightly less code than 13 out of 20 applications (65 percent) and on 8 of these applications (40 percent) it discovered more faults. Concerning the requirements, \( \text{MORE}+ \) covered slightly more requirements on 11 out of 20 applications (55 percent) and on 7 of them (35 percent) it discovered more faults. Even if these manual inspections were not supported by statistical evidences (e.g., for what concerns \( \text{NSGAII}_{2d} \) and \( \text{MORE}+ \)), 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, \( \text{MORE}+ \) 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, \( \text{MORE}+ \) 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 \( \text{MORE}+ \) and baselines). These values indicate the time to obtain \( S_{\text{red}} \), expressed in seconds. Results suggest that on average Cost_Red varied from a few seconds to a few minutes. \( \text{2OPT} \) required more time than other approaches. We can assume that this was due to fact that \( \text{2OPT} \) 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), \( \text{2OPT} \) required an execution time notably greater than other approaches (more than 30 seconds for \( \text{2OPT} \) versus less than 3 seconds for other approaches). However, \( \text{MORE}+ \) 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.
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 section).

5.7 Cost-Effectiveness
To compute CostEff values, we considered 3 subsequent versions of each experimental object (i.e., \( g = 3 \)) in Eqs. (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).

### TABLE 9
Summary of Statistics for \( \text{Cost}_\text{Red} \)

<table>
<thead>
<tr>
<th>Algorithm Execution Time (in seconds)</th>
<th>Min</th>
<th>Median</th>
<th>Mean</th>
<th>Max</th>
<th>sd</th>
</tr>
</thead>
<tbody>
<tr>
<td>MORE+</td>
<td>0.0003</td>
<td>0.04</td>
<td>0.08</td>
<td>0.51</td>
<td>0.12</td>
</tr>
<tr>
<td>2OPT</td>
<td>0.05</td>
<td>3</td>
<td>48</td>
<td>720.5</td>
<td>159.8</td>
</tr>
<tr>
<td>DGF</td>
<td>0.003</td>
<td>0.17</td>
<td>0.25</td>
<td>0.76</td>
<td>0.22</td>
</tr>
<tr>
<td>GRD</td>
<td>0.009</td>
<td>0.41</td>
<td>0.77</td>
<td>3.38</td>
<td>0.8</td>
</tr>
<tr>
<td>HGS</td>
<td>0.0025</td>
<td>0.03</td>
<td>0.039</td>
<td>0.11</td>
<td>0.03</td>
</tr>
<tr>
<td>NSGAII (_{sd})</td>
<td>0.0003</td>
<td>0.04</td>
<td>0.09</td>
<td>0.8</td>
<td>0.18</td>
</tr>
<tr>
<td>GR3D</td>
<td>0.0005</td>
<td>0.33</td>
<td>0.94</td>
<td>5.2</td>
<td>1.33</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Similarity Links Recovering (in seconds)</th>
<th>Reqs - TestCases</th>
<th>Reqs - Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>Median</td>
<td>82.5</td>
<td>19</td>
</tr>
<tr>
<td>Mean</td>
<td>165.6</td>
<td>50.9</td>
</tr>
<tr>
<td>Max</td>
<td>1505</td>
<td>430</td>
</tr>
<tr>
<td>sd</td>
<td>328.6</td>
<td>93.7</td>
</tr>
</tbody>
</table>

### TABLE 10
Cost\(_{Red}\) Values for Experimental Objects

<table>
<thead>
<tr>
<th>APP</th>
<th>Algorithm Execution Time (seconds)</th>
<th>Similarity Links Recovering (seconds)</th>
<th>Total cost</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MORE+ (mean) 2OPT DGR HGS NSGAII (_{sd}) GR3D</td>
<td>MORE+ (mean)</td>
<td></td>
</tr>
<tr>
<td>AveCalc</td>
<td>0.002 0.08 0.05 0.03 0.003 0.002 0.09</td>
<td>2 10</td>
<td>2.09 12</td>
</tr>
<tr>
<td>CommonsBcel</td>
<td>0.02 0.15 0.44 0.09 0.02 0.02 0.11</td>
<td>27 25</td>
<td>27.1 52</td>
</tr>
<tr>
<td>CommonsBeanUtils</td>
<td>0.25 2.18 36.09 0.76 0.03 0.17 3.24</td>
<td>220 10</td>
<td>232.3 230.5</td>
</tr>
<tr>
<td>CommonsCodec</td>
<td>0.04 0.18 6.25 0.13 0.02 0.04 0.26</td>
<td>75 10</td>
<td>75.2 85</td>
</tr>
<tr>
<td>CommonsCollections4</td>
<td>0.07 0.81 30.93 0.38 0.03 0.09 2.23</td>
<td>160 80</td>
<td>162.2 240</td>
</tr>
<tr>
<td>CommonsIOD</td>
<td>0.09 0.83 55.11 0.34 0.04 0.08 0.81</td>
<td>130 30</td>
<td>130.8 160</td>
</tr>
<tr>
<td>CommonsLang</td>
<td>0.52 3.38 720.50 0.72 0.12 0.80 5.27</td>
<td>1505 74</td>
<td>1,510.2 1,579.5</td>
</tr>
<tr>
<td>CommonsProxy</td>
<td>0.00 0.12 0.07 0.04 0.06 0.002 0.01</td>
<td>6 28</td>
<td>6 34</td>
</tr>
<tr>
<td>DbUtils</td>
<td>0.01 0.14 0.50 0.12 0.01 0.01 0.06</td>
<td>13 10</td>
<td>13 23</td>
</tr>
<tr>
<td>iTrust</td>
<td>0.05 0.26 7.55 0.15 0.01 0.05 0.26</td>
<td>120 11</td>
<td>120 131</td>
</tr>
<tr>
<td>Javadoc</td>
<td>0.07 0.77 3.64 0.45 0.04 0.07 1.03</td>
<td>200 104</td>
<td>201 304</td>
</tr>
<tr>
<td>Jtidy</td>
<td>0.01 0.05 0.23 0.02 0.01 0.01 0.01</td>
<td>20 13</td>
<td>20 33</td>
</tr>
<tr>
<td>JXPath</td>
<td>0.05 0.48 1.81 0.20 0.03 0.05 0.40</td>
<td>36 35</td>
<td>36.4 71</td>
</tr>
<tr>
<td>LaTazza</td>
<td>0.0003 0.0092 0.1537 0.0030 0.0026 0.0003 0.0005</td>
<td>3 10</td>
<td>3 13</td>
</tr>
<tr>
<td>Log4j</td>
<td>0.07 1.77 9.34 0.50 0.04 0.07 1.20</td>
<td>90 12</td>
<td>91.2 102</td>
</tr>
<tr>
<td>PMd</td>
<td>0.20 1.81 91.34 0.42 0.06 0.30 2.05</td>
<td>300 75</td>
<td>302 375.2</td>
</tr>
<tr>
<td>Woden</td>
<td>0.04 1.14 2.39 0.20 0.02 0.04 0.63</td>
<td>90 10</td>
<td>90 100</td>
</tr>
<tr>
<td>Xerces</td>
<td>0.17 0.77 6.79 0.31 0.05 0.16 0.75</td>
<td>27 10</td>
<td>27.2 37</td>
</tr>
<tr>
<td>xmlGraphics</td>
<td>0.02 0.27 0.90 0.11 0.02 0.02 0.26</td>
<td>15 31</td>
<td>15.2 46</td>
</tr>
<tr>
<td>xmlSecurity</td>
<td>0.02 0.36 1.70 0.11 0.10 0.02 0.24</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### TABLE 11
CostEff: Statistics on the 20 Applications

<table>
<thead>
<tr>
<th>App.#</th>
<th>Release</th>
<th>MORE+</th>
<th>2OPT</th>
<th>DGR</th>
<th>HGS</th>
<th>NSGAII (_{sd})</th>
<th>GR3D</th>
<th>Mean CostEff</th>
<th>Std.dev CostEff</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1,254</td>
<td>1,262</td>
<td>1,214</td>
<td>1,268</td>
<td>1,261</td>
<td>1,274</td>
<td>1,446</td>
<td>674.1</td>
<td>788.8</td>
</tr>
<tr>
<td>2</td>
<td>2,291</td>
<td>2,475</td>
<td>2,427</td>
<td>2,535</td>
<td>2,522</td>
<td>2,331</td>
<td>2,725</td>
<td>1,089.2</td>
<td>1,342.7</td>
</tr>
<tr>
<td>3</td>
<td>3,328</td>
<td>3,689</td>
<td>3,640</td>
<td>3,802</td>
<td>3,783</td>
<td>3,388</td>
<td>4,005</td>
<td>1,523.2</td>
<td>1,910.5</td>
</tr>
</tbody>
</table>
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 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 versus MORE+/traditional approaches (e.g., 2OPT, GRD), as well as descriptive statistics and statistical tests. By comparing Table 12 with results shown in Tables 4 and 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 $A_{12}$ ($>0.19$). 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 percent), 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 the smaller size, in 7 out of such 9 applications (77 percent), 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.

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 Fig. 4), and the number of faults discovered by each suite per experimental object for both MORE+ (see the blue boxplots in Fig. 4) and $NGSAI_{2d}$ (see the white boxplots in Fig. 4). We recall that Table 13 reports the mean values of discovered faults for MORE+, $NGSAI_{2d}$, and traditional approaches.

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

- Group (A): applications for which almost all suites (more than 90 percent) in the Pareto fronts were
more effective than the mean trend of traditional approaches: 9 for MORE+ and 7 for NSGAII<sub>2d</sub>.

- Group (B): applications for which a large amount (between 65 and 90 percent) of suites in the fronts were more effective than the mean trend of traditional approaches: 5 for MORE+ and 4 for NSGAII<sub>2d</sub>.

- Group (C): applications for which in a range between 50 and 65 percent of suites in the fronts were more effective than the mean trend of traditional approaches: 1 for MORE+ and 2 for NSGAII<sub>2d</sub>.

- Group (D): applications for which almost all (more then 90 percent) of suites in the fronts were less effective than the mean trend of traditional approaches: 1 for MORE+ and 2 for NSGAII<sub>2d</sub>.

- Group (E): applications for which a large amount (between 65 and 90 percent) of suites in the fronts were less effective than the mean trend of traditional approaches: 3 for MORE+ and 2 for NSGAII<sub>2d</sub>.

- Group (F): applications for which in a range between 50 and 65 percent of suites in the fronts were less effective than the mean trend of traditional approaches: 1 for MORE+ and 2 for NSGAII<sub>2d</sub>.

### TABLE 13
F (Faults) versus Suite Size: Comparison Between Traditional Approaches, MORE+ and NSGAII<sub>2d</sub>

<table>
<thead>
<tr>
<th>Suite Size (mean)</th>
<th>Faults (mean)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Traditional</td>
</tr>
<tr>
<td>AveCalc</td>
<td>6</td>
</tr>
<tr>
<td>CommonsBcel</td>
<td>46.6</td>
</tr>
<tr>
<td>CommonsBeanUtils</td>
<td>234.2</td>
</tr>
<tr>
<td>CommonsCodec</td>
<td>95.8</td>
</tr>
<tr>
<td>CommonsCollections4</td>
<td>322.8</td>
</tr>
<tr>
<td>CommonsIO</td>
<td>389.6</td>
</tr>
<tr>
<td>CommonsLang</td>
<td>96.8</td>
</tr>
<tr>
<td>CommonsProxy</td>
<td>38.4</td>
</tr>
<tr>
<td>DbUtils</td>
<td>49.4</td>
</tr>
<tr>
<td>iTrust</td>
<td>170</td>
</tr>
<tr>
<td>Jabref</td>
<td>91</td>
</tr>
<tr>
<td>Jtidy</td>
<td>29</td>
</tr>
<tr>
<td>JXPath</td>
<td>73</td>
</tr>
<tr>
<td>LaTazza</td>
<td>1.2</td>
</tr>
<tr>
<td>Log4j</td>
<td>285</td>
</tr>
<tr>
<td>Pmd</td>
<td>100.4</td>
</tr>
<tr>
<td>Woden</td>
<td>107.8</td>
</tr>
<tr>
<td>Xerces</td>
<td>24</td>
</tr>
<tr>
<td>xmlGraphics</td>
<td>50.4</td>
</tr>
<tr>
<td>xmlSecurity</td>
<td>36</td>
</tr>
</tbody>
</table>

Fig. 4. Box-plots of faults discovered by all solutions in Pareto fronts of MORE+ (blue filled boxplots) and NSGAII<sub>2d</sub> (white boxplots), mean effectiveness of traditional approaches (continuous black line), and their groups (letters from A to F) for MORE+ and NSGAII<sub>2d</sub>.
These groups indicated that on 18 out of 20 applications for MORE+ and 15 out 20 for NSGAII, 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 NSGAII 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 NSGAII 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 lead to different results.

Furthermore, boxplots for MORE+ and NSGAII in Fig. 4 graphically expressed the statistical data already shown in Table 4. Such data suggest that MORE+ discovered significantly more faults than NSGAII 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 NSGAII. 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 255), in the fronts for which traditional approaches achieved better results (i.e., Eff(MORE+) < Eff(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 (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 NSGAII. 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 NSGAII that had higher effectiveness than the suites reduced by traditional approaches, 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 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 percent of suites reduced by MORE+ were not dominated by suites reduced by traditional approaches (for all 20 applications, 100 percent of suites reduced by MORE+ were not dominated). Conversely, only 26 percent of suites reduced by traditional approaches were not dominated. We measured the Hypervolume (HV) metric [63], [78] of each reference front and compared them by applying the Mann-Whitney test and the \( A_{12} \) 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 \( A_{12} \) (\( A_{12} \) = 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 percent of suites reduced by NSGAII\(_{2d}\) were not dominated by suites reduced by traditional approaches (for all 20 applications, 100 percent of suites reduced by NSGAII\(_{2d}\) were not dominated). Conversely, only 28 percent 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 \( A_{12} \) value was 0.75. Results suggest 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

### TABLE 15

<table>
<thead>
<tr>
<th>App</th>
<th>(HV_{\text{MORE+}})</th>
<th>(HV_{\text{NSGAII}_{2d}})</th>
<th>(HV_{\text{MORE+}})</th>
<th>(HV_{\text{NSGAII}_{2d}})</th>
</tr>
</thead>
<tbody>
<tr>
<td>AveCalc</td>
<td>0.0</td>
<td>0.2</td>
<td>7.3</td>
<td>0.0</td>
</tr>
<tr>
<td>CommonsBcel</td>
<td>2,198.6</td>
<td>249.6</td>
<td>3,827.9</td>
<td>901.1</td>
</tr>
<tr>
<td>CommonsBeanUtils</td>
<td>139,976.7</td>
<td>58.8</td>
<td>140,045.0</td>
<td>139,976.7</td>
</tr>
<tr>
<td>CommonsCodec</td>
<td>9,484.7</td>
<td>2,414.9</td>
<td>234.2</td>
<td>131.6</td>
</tr>
<tr>
<td>CommonsCollections4</td>
<td>36,777.6</td>
<td>2,445.4</td>
<td>4,467.4</td>
<td>4,467.4</td>
</tr>
<tr>
<td>CommonsIO</td>
<td>8,678.4</td>
<td>5,710.0</td>
<td>8,154.5</td>
<td>345.6</td>
</tr>
<tr>
<td>CommonsLang</td>
<td>179,978.5</td>
<td>14,976.5</td>
<td>52,878.3</td>
<td>1,113.3</td>
</tr>
<tr>
<td>CommonsProxy</td>
<td>377.6</td>
<td>0.0</td>
<td>225.1</td>
<td>225.1</td>
</tr>
<tr>
<td>DbUtils</td>
<td>1,092.1</td>
<td>807.1</td>
<td>499.0</td>
<td>30.4</td>
</tr>
<tr>
<td>iTrust</td>
<td>107.0</td>
<td>92,016.1</td>
<td>12,290.8</td>
<td>107.0</td>
</tr>
<tr>
<td>Jabref</td>
<td>155,385.2</td>
<td>25,554.1</td>
<td>106,791.2</td>
<td>2,527.6</td>
</tr>
<tr>
<td>Jtidy</td>
<td>1.6</td>
<td>230.9</td>
<td>206.4</td>
<td>1.6</td>
</tr>
<tr>
<td>JXPath</td>
<td>7,414.9</td>
<td>1,349.3</td>
<td>1,415.8</td>
<td>1,321.7</td>
</tr>
<tr>
<td>Latazza</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Log4j</td>
<td>208,429.7</td>
<td>2,502.7</td>
<td>131,177.1</td>
<td>2,838.3</td>
</tr>
<tr>
<td>Pmd</td>
<td>2,352.6</td>
<td>481.7</td>
<td>2,509.8</td>
<td>2,352.6</td>
</tr>
<tr>
<td>Woden</td>
<td>7,454.4</td>
<td>1,220.6</td>
<td>4,289.1</td>
<td>105.3</td>
</tr>
<tr>
<td>Xeres</td>
<td>810.2</td>
<td>196.9</td>
<td>1,036.4</td>
<td>810.2</td>
</tr>
<tr>
<td>xmxGraphics</td>
<td>38,490.1</td>
<td>16,133.5</td>
<td>40,681.6</td>
<td>16,305.8</td>
</tr>
<tr>
<td>xmlSecurity</td>
<td>5,708.9</td>
<td>2,042.1</td>
<td>44,192.3</td>
<td>4,490.5</td>
</tr>
</tbody>
</table>

\(p - value\) 0.08 0.01 0.05
\(A_{12}\) 0.34 0.75 0.68
were mainly not dominated (Mann-Whitney \( ^* \) test). On the obtained HV (17 out of 20 applications, 100 percent of suites were not dominated). Conversely, only 66 percent of suites reduced by NSGAII were not dominated (only for 8 applications, 100 percent of suites reduced by NSGAII were not dominated). The obtained HV values (see Table 15), we computed the Mann-Whitney test and \( A_{12} \). The obtained values were 0.05 and 0.6825, respectively. Results show that by additionally considering suites generated by NSGAII, 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 percent (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 revealed at least one fault. For example, 3 out of 20 faults (15 percent) of CommonsCollections did not impact on the set of considered requirements. In the fourth column of Table 16, we report fault density (\( \text{FaultDensity} = \frac{\sum_{r \in \text{Reqs}} (\frac{\text{NumFaults}}{\text{NumReqs}} - \text{median} (\frac{\text{NumFaults}}{\text{NumReqs}})))}{\text{NumReqs}} \)). 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 percent 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: AveCalc, CommonsProxy, DbUtils, iTrust, CommonsBeanUtils, and Xerces. Conversely, the faults were concentrated in a few requirements (i.e., distributed in less than 51 percent 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 DbUUtils (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 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 percent 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 percent of fault-revealing links used in the empirical study could be incomplete.
fault, respectively. We can observe that only in AveCalc and LaTazza a large percentage of test cases (about 50 percent) 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. The average less than 8 percent of test cases revealed only one fault, LaTazza had the highest percentage 17 percent, while 10 applications had less than 5 percent 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: \( \text{Reps} \). TCSRedundancy is the set of test cases 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.

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 \( \geq 1 \) fault 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 relatively 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 percent for RF, 52.4 percent for RS, and 69.8 percent for RT). In particular, 2OPT, GRD and GR3D had the highest variability for RF (94.7 percent versus an average of 85.6 percent of other approaches), while MORE+ had the highest variability for RS (64 percent versus an average of 50.5 percent of other approaches) and RT (78.7 percent versus an average of 68.3 percent 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 \( geq 1 \) fault 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 \( geq 1 \) fault 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 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 that 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 suites that cover less application code.

<table>
<thead>
<tr>
<th>Co-factor</th>
<th>Correlation</th>
<th>RF</th>
<th>RS</th>
<th>RT</th>
</tr>
</thead>
<tbody>
<tr>
<td>MORE+</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reqs</td>
<td>-17%</td>
<td>-2%</td>
<td>-7%</td>
<td></td>
</tr>
<tr>
<td>NotTestedReqs</td>
<td>-19%</td>
<td>9%</td>
<td>-13%</td>
<td></td>
</tr>
<tr>
<td>TCSRedundancy</td>
<td>-11%</td>
<td>34%</td>
<td>10%</td>
<td></td>
</tr>
<tr>
<td>TCSRevealing ( \geq 1 ) fault</td>
<td>-11%</td>
<td>-3%</td>
<td>-20%</td>
<td></td>
</tr>
<tr>
<td>TCSRevealing1fault</td>
<td>-4%</td>
<td>-5%</td>
<td>-22%</td>
<td></td>
</tr>
<tr>
<td>GR3D</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reqs</td>
<td>-23%</td>
<td>-22%</td>
<td>-28%</td>
<td></td>
</tr>
<tr>
<td>NotTestedReqs</td>
<td>-35%</td>
<td>5%</td>
<td>-15%</td>
<td></td>
</tr>
<tr>
<td>TCSRedundancy</td>
<td>-23%</td>
<td>5</td>
<td>-15%</td>
<td></td>
</tr>
<tr>
<td>TCSRevealing ( \geq 1 ) fault</td>
<td>-33%</td>
<td>-15%</td>
<td>-30%</td>
<td></td>
</tr>
<tr>
<td>TCSRevealing1fault</td>
<td>-23%</td>
<td>-19%</td>
<td>-33%</td>
<td></td>
</tr>
<tr>
<td>Other approaches</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reqs</td>
<td>15%</td>
<td>22%</td>
<td>15%</td>
<td></td>
</tr>
<tr>
<td>NotTestedReqs</td>
<td>-11%</td>
<td>23%</td>
<td>4%</td>
<td></td>
</tr>
<tr>
<td>TCSRedundancy</td>
<td>-18%</td>
<td>38%</td>
<td>11%</td>
<td></td>
</tr>
<tr>
<td>TCSRevealing ( \geq 1 ) fault</td>
<td>4%</td>
<td>27%</td>
<td>18%</td>
<td></td>
</tr>
<tr>
<td>TCSRevealing1fault</td>
<td>23%</td>
<td>21%</td>
<td>17%</td>
<td></td>
</tr>
</tbody>
</table>

**Bold values are statistically significant at 95 percent.**
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 un-reduced 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 artifacts 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.

Acknowledgments
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


Alessandro Marchetto received the PhD degree in software engineering from the University of Milano, Italy, in 2007. He is currently an independent researcher working in the field of Software Engineering. From 2006 till the end of 2012, he was a researcher in 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., the *IEEE Transactions on Software Engineering*, the *Information and Software Technology*, the *Journal of Systems and Software*, the *Empirical Software Engineering*, the *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 the Laurea and PhD degrees, both in computer science, from the University of Salerno, Italy, in 2001 and 2003, respectively. In 2006, he joined, as an assistant professor in the Department of Mathematics and Computer Science, University of Basilicata, Potenza, Italy. In 2015, he became an associate professor with the University of Basilicata. 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 refereed papers in journals, books, and conference proceedings. He serves on the organizing committees 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). He leads both the Group and the Laboratory of Software Engineering, University of Basilicata. He has published more than 150 refereed papers in journals, books, and conference proceedings. He serves on the organizing committees 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). He is a member of the IEEE and the IEEE Computer Society. More on http://www2.unibas.it/gscanniello/

Angelo Susi is a research scientist in the Software Engineering Group, Fondazione Bruno Kessler in Trento, Italy. His research interests include 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 the *IEEE Transactions on Software Engineering*, the *ACM Transactions on Software Engineering and Methodology*, the *Information and Software Technology*, the *Software and Systems Modeling*, *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 the *IEEE Transactions on Software Engineering*, the *Romanian Economic Journal*, the *Information and Software Technology*, the *Journal of Systems and Software*. He has been the scientific manager of the EU FP7 project RISCOSS. He is a member of the IEEE.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.