A Multi-Objective Technique to Prioritize Test Cases

Alessandro Marchetto, Md. Mahfuzul Islam, Waseem Asghar, Angelo Susi, Giuseppe Scanniello, *Member, IEEE*

Abstract—While performing regression testing, an appropriate choice for test case ordering allows the tester to early discover faults in source code. To this end, test case prioritization techniques can be used. Several existing test case prioritization techniques leave out the execution cost of test cases and exploit a single objective function (e.g., code or requirements coverage). In this paper, we present a multi-objective test case prioritization technique that determines the ordering of test cases that maximize the number of discovered faults that are both technical and business critical. In other words, our new technique aims at both early discovering faults and reducing the execution cost of test cases. To this end, we automatically recover links among software artifacts (i.e., requirements specifications, test cases, and source code) and apply a metric-based approach to automatically identify critical and fault-prone portions of software artifacts, thus becoming able to give them more importance during test case prioritization. We experimentally evaluated our technique on 21 Java applications. The obtained results support our hypotheses on efficiency and effectiveness of our new technique and on the use of automatic artifacts analysis and weighting in test case prioritization.

Index Terms—Regression Testing; Requirements; Testing; Test Case Prioritization.

1 Introduction

The intent of regression testing is to ensure that enhancements, patches, or configuration changes have not introduced new faults in source code. Relevant activities in regression testing are [1]: *(i)* test case selection; *(ii)* test case minimization; and *(iii)* test case prioritization. The goal of test case selection is to choose test cases that are relevant for a specific part of an application or for performed changes. On the other hand, test case minimization aims at reducing the number of test cases to be executed by removing redundant test cases, thus preserving the capability of a test case suite in discovering faults. Finally, the goal of test case prioritization is to determine test case ordering that maximizes the probability to early discover faults in source code. In other words, it is of primary importance to identify test case orderings that are effective (in terms of capability in early discovering faults) and efficient (in terms of execution cost). These factors are relevant because they represent technical and business criteria for the success of a software project [2].

Test case prioritization techniques [1], [3] exploit several algorithms to prioritize test cases. These techniques are mostly based on a single dimension (e.g., code or requirements coverage) and assume that faults have all the same relevance and that all software artifacts (e.g., source code and requirements) are equally relevant. That is, these techniques do not identify test case orderings that early reveal both technical (e.g., coding faults) and business critical faults (e.g., due to the misunderstanding of requirements).

We presented in [4] a technique to prioritize test cases that explicitly considers: low- and high-level information about test cases. In particular, it was based on the three following dimensions: *structural* that concerns information on source code exercised by test cases under analysis; *functional* that regards coverage of users' and application requirements; and *cost* that concerns time to execute test cases. A test case ordering was attained as a multi-objective optimization problem to balance considered dimensions with respect to traceability links among software artifacts (i.e., application code, test cases, and requirements specifications). These links were recovered by applying Latent Semantic Indexing (LSI) [5]. It is an established Information Retrieval (IR) technique largely exploited to recover traceability links (e.g., [6], [7]). A limitation for our previous presented technique [4] is that it equally weighted all portions of application artifacts (i.e., source code and requirements) during test case prioritization. However, it is often the case in which different portions of application artifacts have different fault-proneness or testers have specific needs (e.g., a given requirement or function has to be tested first). To overcome that limitation, testers could be asked to manually identify critical portions of application artifacts, thus becoming able to give them more importance during test case prioritization. However, this approach is costly for human testers and also error-prone. In this paper, we improve the solution highlighted before by leveraging the capability of automatically identifying fault-prone portions of software artifacts, according to some characteristics of

- *A. Marchetto, M. M. Islam, and W. Asghar are Independent researchers E-mail: {alex.marchetto, mahfuzul.islam, waseem960}@gmail.com*
- *A. Susi is with Fondazione Bruno Kessler E-mail: susi@fbk.eu*
- *G. Scanniello is with DiMIE - University of Basilicata. E-mail: giuseppe.scanniello@unibas.it*

the source code of a given application (e.g., McCabe Cyclomatic Complexity) and its requirements (e.g., the number of classes that implement a requirement). Summarizing our approach provides the following new research contributions: *(i)* a novel multi-objective test case prioritization technique; *(ii)* the definition of a metric-based approach to automatically identify potential critical and fault-prone portions of application code and requirements; and *(iii)* a large experimental evaluation.

As for our experimental evaluation, we have conducted an experiment on 21 Java applications. We also compared results obtained by applying our technique and those by baseline approaches for test case prioritization, namely random prioritization, code and additional code coverage techniques, and a multi-objective approach based on code coverage and test case execution cost [12], [13], [14], [18]. Another baseline technique for comparison was that we previously presented in [4]. Outcomes suggested that our technique is able to identify test case orderings that are effective in terms of early fault discovery and efficient in terms of test case execution cost.

In Section 2, we discuss related work, while the outline of our approach is given in Section 3. The approach exploited to recover traceability links is introduced in Section 4. In Section 5, we present metrics and measurements used in our technique, which is successively shown in Section 6. In Section 7, we summarize the design of our investigation and present and discuss achieved results. Final remarks conclude the paper.

2 Related Work

To prioritize and select test cases a number of techniques have been proposed and empirically investigated [8], [9], [10], [11], [12], [13], [14], [15], [16]. Yoo *et al.* [1] and Mohanty *et al.* [3] survey existing research work in these fields. Results suggest that existing techniques mostly use either structural or functional coverage criteria with respect to source code executed by test cases. This is one of the aspects that makes our proposal different from those in the literature.

A number of approaches use code coverage and additional code coverage[1] to prioritize test cases with respect to their capability of executing the source code of software under test (e.g., [14], [17]). Most of these approaches identify test case orderings based on a single objective function (e.g., code coverage). Only a few approaches based on multi-objective optimization exist (e.g., [8], [18]). These approaches mainly consider code coverage information and execution cost of test cases: *(i)* optimize test cases by means of a Pareto front using both code coverage and execution cost or *(ii)* reduce a multi-objective problem to a single-objective by using an optimization function. For example, Yoo and Harman [18] show same benefits of a Pareto-front optimality for test case selection. The authors present a two-objective test case selection approach, where code coverage and execution cost are explicitly considered when conducting test case selection. The approach can be also directly applied to test case prioritization. This work presents some similarities with that we present in this paper, namely the objective formulation takes into account source code coverage as a measure of test adequacy and execution time as a measure for cost. The most remarkable differences between these two approaches can be summarized as follows: we also consider the coverage of application requirements, to link them with source code we applied an IR technique, and we apply a metric-based approach to automatically identify critical and fault-prone portions of software artifacts (both source code and requirements). Another multi-objective test case prioritization approach is proposed by Sun *et al.* [19] for ordering test cases in GUI-based applications. In fact, code (statement) coverage is traditionally used to test case prioritize, while event coverage criteria are largely adopted for GUI applications testing [20]. Hence, Sun *et al.* propose a multi-objective test case prioritization approach that exploits both criteria: statement and event coverage.

More traditionally, Salehie *et al.* [9], Kavitha *et al.* [21], Arafeen *et al.* [22], and Nguyen *et al.* [23] propose techniques to prioritize test cases according to application requirements. Test cases are mapped to requirements using a text-to-text traceability links recovery technique and then test cases are prioritized with the aim of maximizing user satisfaction. In contrast with our proposal, the most critical aspect of such techniques is that they mainly prioritize test cases according to the sole information coming from requirements, so ignoring the structure and the behavior of application under test.

Yoo *et al.* [11] propose an approach to prioritize test cases according to tester's needs, while considering structural information of software under test. Authors ask testers to prioritize test cases conducting a pair-wise comparison of them. To limit human effort, authors combine this manual pair-wise comparison of test cases with test clustering based on coverage information, thus improving scalability of their technique. Then, testers are asked to prioritize groups of test cases (according to the group representative test case) rather than every test case. The authors also assume that testers have complete knowledge on each test case. This could not be true, e.g., in functional testing or in case of a huge test suite.

Walcott *et al.* [10] present a technique to prioritize test cases with respect to time constraints. This is typical for those contexts in which execution time is limited by environment constraints. This technique achieves good results in terms of effectiveness. However, a few assumptions are taken: different types of faults have same severity and execution cost of every test case is uniform. These assumptions might be true only in a few specific contexts (e.g., applications based on the composition of third-party services).

Fang *et al.* [24] propose a similarity-based technique that uses execution profiles of test cases to maximize diversity of test cases. The execution frequency profiles of test cases

1. Additional code coverage techniques evaluate each test case according to the code portion that is uniquely covered by it

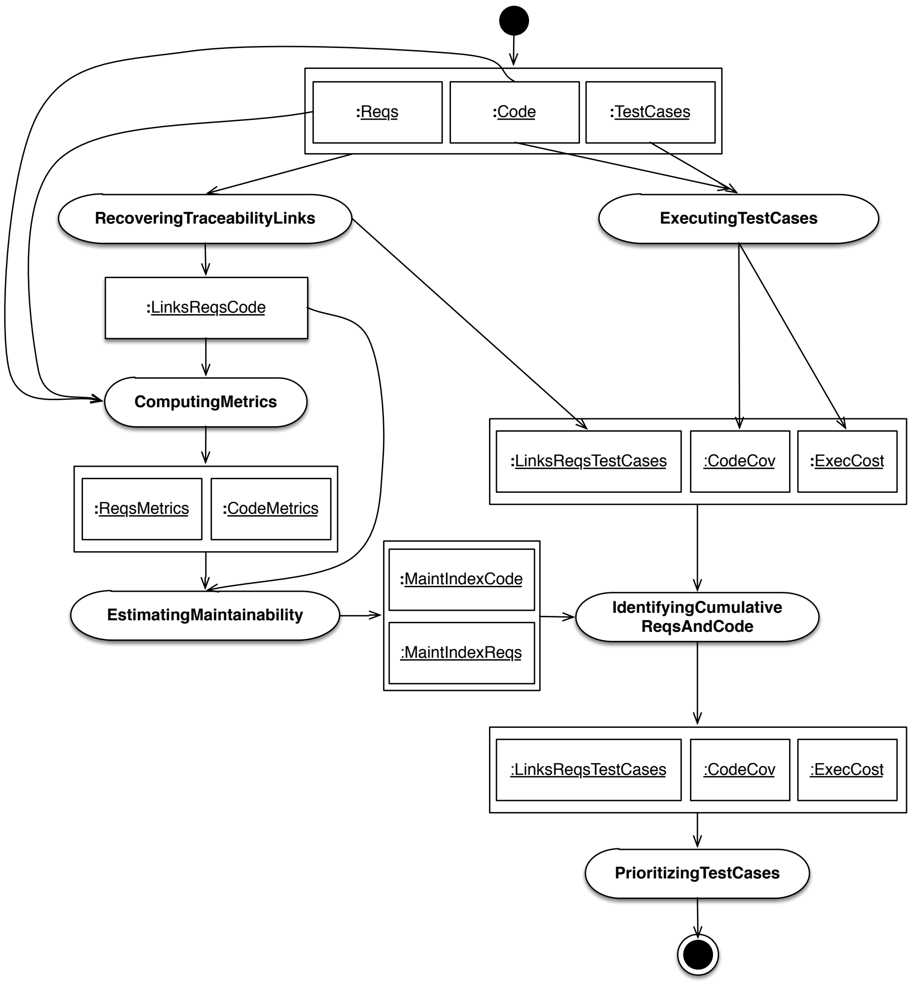


Fig. 1. Process outline modelle with an UML Activity Diagram with Object Flow

are collected and transformed into ordered sequences. Then, test case diversity is computed by applying string edit distances between each pair of execution sequences of test cases. This dis/similarity measure is used to establish test case prioritization.

Li *et al.* [8] empirically assess effectiveness of greedy and meta-heuristic algorithms to prioritize test cases using code coverage measures. Conversely to other works, that focused on the best criteria (e.g., code coverage, time) to prioritize test cases, Li *et al.* mainly focus on the algorithm used to compute optimal test case orderings. Results suggest that meta-heuristic algorithms seem to be quite efficient and effective for traversing the solution space, thus promising to define optimal test orderings. According to these results, we propose in our work the use of a meta-heuristic algorithm to prioritize test cases according to the three considered dimensions.

Unlike the studies discussed before, we propose a technique to prioritize test cases that considers low- (e.g., code coverage) and high-level (e.g., requirements coverage) information about test cases and that uses automatically recovered traceability links among requirements, source code, and test cases. Another remarkable difference between our work and those in the literature is that we have conducted a more extensive experimentation on several test suites and a large number of applications.

3 Approach Outline

We present a multi-objective test case prioritization technique that determines the ordering of test cases that maximize the number of discovered faults that are both technical and business critical. This approach automatically recovers traceability links among software artifacts and applies a metric-based approach to automatically identify critical and fault-prone portions of software artifacts. In Figure 1, we show a behavioral view of our approach in terms of an UML Activity Diagram with object flow [25]. Ellipses are phases of our process, while rectangles are software artifacts produced and/or consumed in each phase.

For each test case, the ExecutingTestCase phase provides details on covered code statements and execution cost (the artifacts :CodeCov and :ExecCost in Figure 1), namely two of the dimensions on which our approach is based on. The RecoveringTraceabilityLinks phase is in charge of recovering links between requirements and source code (:LinksReqsCode) and between requirements and JUnit test

```
package calculator;
import java.io.Serializable;
import javax.swing.JOptionPane;
/**
 * Classe Exam.
 * Bean representing an exam.
 *
 * @author   andima
 */

public class Exam implements Serializable \{
    public String name;
    public int cfu;
    public int vote;
    public boolean laude;
    public boolean maked;
    public Exam()\{
        name = "Unknow";
        cfu = -1;
        vote = -1;
        laude = false;
        maked = false;
    \}
    public static Exam getInstance(String name,
            String cfu, String vote)\{
        Exam e = new Exam();
        e.setName(name);
        // ...
    \}
    // ...
\}
```

Fig. 2. The class `calculator.Exam` in AveCalc

```
A student can add a new exam to the
register. An exam is composed of a name,
CFU (i.e., a number that represent the
university credit of the exam) and an
optional vote. The name is unique, CFU
is a positive number (>=0) and the vote,
if inserted, is a number included between
0 and 30 (the vote can be also 0 or 30).
A vote < 18 is negative (i.e., the exam
is not passed) while >= 18 is positive
(i.e., the exam is passed). An exam can be
inserted also without the vote; it can be
inserted later. 'Laude' can be added only
when the vote is 30.
```

Fig. 3. The requirement `AddExam` in AveCalc

cases (:LinksReqsTestCases). Source code is considered as text and also requirements since they are described in natural language. In Figure 2 and Figure 3, we show a fragment of the class `calculator.Exam` and requirement `addExam` of the application AveCalc[2], respectively. LSI allowed us to find a link between `calculator.Exam` and `addExam`. Links between requirements and classes estimate the coverage of requirements that represents the third dimension of our approach. We provide details on the approach used for the recovery of traceability links in Section 4.

Source code and requirements metrics (:CodeMetrics and :ReqsMetric, respectively) are computed in the ComputingMetrics phase. Traceability links between requirements and source code are also used to compute requirements-level metrics. These links and both source code and requirements metrics are then used to estimate maintainability indexes (:MaintIndexCode and :MaintIndexReqs) for the classes and the requirements of a given subject software in the phase EstimatingMaintainability. Requirements and source code classes are ordered according to their maintainability indexes, respectively. These orderings are then used together with covered code statements and execution costs (i.e., the output of ExecutingTestCase) and recovered traceability links between requirements and test cases to compute cumulative measures for traceability links and both code coverage and execution costs of test cases. These measures are exploited to identify portions of application code and requirements that are potentially critical and fault-prone in the phase IdentifyingCumulativeReqsAndCode. In Section 5, we describe the three phases of our approach we described just before. It is worth remarking that the support provided by these three phases represents the most important difference between our current contribution and that we previously presented [4], where testers had to manually identify critical portions of application artifacts (e.g., source code and requirements).

The performance evaluation of all possible test case orderings on the three choose dimensions is expensive in case of test suites containing a large number of test cases. To deal with this issue, the PrioritizingTestCase phase exploits a multi-objective optimization method to prioritize test cases according to our three dimensions. Several possible evolutionary algorithms are available and applicable to the problem of test case prioritization. In the work presented here, we rely on the Non-dominated Sorting Genetic Algorithm II (NSGA-II [26]). In fact, NSGA-II is widely used in the solution of optimization problems in software engineering and demonstrated to be particularly suited for the prioritization problem [27], [28]. In Section 6, we provide details on how NSGA-II has been used in our new approach.

4 Traceability Recovery

Requirements traceability regards the documentation of bi-directional links among various related requirements and associated software artifacts produced in the entire development process. In other words, requirements traceability refers to the ability to describe and follow the life of a requirement, from its origins, through its development and specification, to its subsequent deployment and use, and through all periods of on-going refinement and iteration in any of these phases [29]. This allows a software engineer to understand relationships that exist within and across different kinds of software artifacts. For example, documentation of traceability links might be crucial to be aware about: *(i)* source code in charge of implementing a given application requirement; *(ii)* requirements implemented by a specific part of the source code; and *(iii)* source code exercised by a test case.

Traceability links are very often not documented at all and if this information exists it might be not updated or not aligned with the current implementation and documentation (e.g., [30], [31], [32], [33]). Therefore, methods and

2. It is one of the applications used in our empirical assessment

tools might be needed to infer traceability links among software artifacts and requirements and source code, in particular. In this regard, researchers have successfully applied IR techniques [6], [33], [34], [35]. These approaches are mostly based on lexical similarity of text contained in these artifacts [36]. In particular, artifacts are indexed by extracting information about occurrences of terms within them and then a lexical similarity measure is computed to establish whether or not a traceability link might exist between two artifacts. Independently from the IR technique, the process to recover traceability links among software artifacts is similar.

LSI (sometimes referred to as Latent Semantic Analysis) has been successfully applied in traceability field (e.g., [6], [36]). Other text retrieval and IR techniques have been successfully applied to the problem of recovering traceability links among software artifacts. However, existing research is contradictory on which text retrieval model and technique work best with source code data. For example, Marcus and Maletic [6] experimentally observed that LSI performs at least as well as Vector Space Model (VSM) [37] and in some cases LSI outperforms VSM in recovering documentation-to-source-code traceability links. Conversely, Abadi *et al.* [38] observed that VSM provides better results than LSI in the recovery of traceability links among different kinds of software artifacts. Similar results were also obtained by Wang *et al.* [39]. Other authors advocate for the use of Latent Dirichlet Allocation (LDA) [40]. We decided to use LSI because it is efficient and widely used in traceability recovery field. The used approach is close to that proposed by Marcus and Maletic [6] and then assessed by De Lucia *et al.* [36]. The use of a different text retrieval model would not alter the results of our test case prioritization approach. The use of a different IR technique represent a possible future direction for our research.

4.1 Latent Semantic Indexing

LSI assumes that there is some underlying or latent structure in word usage that is partially obscured by variability in word choice, and uses statistical techniques to estimate this latent structure. LSI uses information about co-occurrence of terms (latent structure) to automatically discover synonymy between two or more terms. The latent structure of the content is obtained by applying a Singular Value Decomposition (SVD) 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). By applying SVD, each term and each artifact could be represented by a vector in the k space (i.e., the dimensionality reduction of the latent structure) of underlying concepts. Indeed, we use SVD to construct a low-rank approximation C_k to the term-document matrix, for a value of k that is far smaller than original rank of C. Thus, we map each row/column to a $k-$dimensional space, which is defined by k principal eigenvectors (corresponding to the largest eigenvalues) of CC^T and C^TC. The matrix C_k is itself still an $m \times n$ matrix, irrespective of k. The selection of an appropriate value for k is an open issue. A value for k should be large enough to fit the real structure of text, but small enough so that we do not also fit the sampling error or unimportant details.

4.2 IR-Based Traceability Recovery

In a typical text retrieval problem, a software engineer writes a textual query and retrieves documents that are similar to that query. In IR-based traceability recovery a set of source artifacts (used as the query) are compared with set of target artifacts (even overlapping). Hence, the number of queries is equal to the number of source artifacts.

To compute similarities between vectors, we use the new k-dimensional space as we did the original representation. Similarity between vectors can be computed by different measures (e.g., Euclidean distance) [41]. In traceability recovery, the widely used measure is cosine similarity [36] between each pair of source and target software artifacts. The larger the cosine similarity value, the more similar the source artifact to the target one is.

Source artifacts are normalized in the same way as target ones (i.e., the corpus). Different set of techniques could be used (e.g., stop word removal and/or stemming). In our case, normalization is performed by removing non-textual tokens, splitting terms composed of two or more words, and eliminating all the terms from a stop word list and with a length less than three characters. Finally, a Porter stemmer [41] is applied on lexemes to reduce them to their root form.

All possible pairs (candidate traceability links) are reported in a ranked list. Irrelevant pairs of artifacts can be removed using a threshold that selects only a subset of top links, i.e., retrieved links. Well known strategies for threshold selection are [36]: *Constant Threshold*, a constant threshold is chosen; *Scale Threshold*, a threshold is computed as percentage of best similarity value between two vectors; *Variable Threshold*, all links among those candidate are retrieved links whether their similarity values are in a fixed interval. In this work, we use the Constant Threshold strategy to limit possibility of loosing links by considering a large number of link candidates. IR-based traceability recovery approaches retrieve also links between source code and target artifacts that do not coincide with correct ones: some are correct and others not. This is why these approaches are semi-automatic and require human intervention to remove erroneously recovered traceability links. To reduce possible biases in test case prioritization results due to human factors/decisions, we do not perform any further analysis to remove erroneously recovered traceability links. It is worth mentioning that a traceability recovery process could be executed (e.g., in background) every time a tester want or requirements and/or source code are modified in accordance to maintenance tasks. In our case, this choice reduces the impact of the overhead computational cost for the recovery of traceability links on the execution of our test case prioritization approach.

5 Relevant Code and Requirements

In the following, we present metrics and algorithms proposed to identify portions of application code and requirements that are potentially critical and fault-prone.

5.1 Metrics

Code. Fault detection capability of a test suite cannot be known before executing test cases. Therefore, we have to resort to potential fault detection capability of a test suite. It can be estimated considering the amount of code covered by test cases in a test suite at run-time [12]. A test case that covers a larger set of code statements has a higher potential fault detection capability (i.e., potentially more faults should be revealed) than one test case that covers a smaller set of statements.

We define *CCov(t)* as the amount of code statements exercised during the execution of a given JUnit test case *t*. A variant of this code coverage measure is *WCCov(t)*. For a given test case, it is defined as a weighted source code coverage measure in which the coverage of source code is computed as follows:

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

where *Statements* is the set of source code statements. *CodeCovered* is the set of statements covered by the execution of the test case *t*, while *s* is a code statement of an application and w_s ($0 \leq w_s \leq 1$) is a predefined weight associated to each code statement. The higher the w_s value, the greater the relevance a tester gives to statements is. In our previous work [4], we left the tester to manually specify such a weight for different parts (e.g., Java classes and packages) of code. In fact, this weight w_s is expected to be useful to customize the measurement of code coverage according to testing needs. For example, a class implementing a critical service for an application needs to be tested more than other classes. In our approach, we exploit a metric-based approach to automatically identify such a weight for each Java class of the application under test by considering code characteristics. Code metrics allow ordering application classes according to their estimated fault-proneness when computing artifact coverage.

Given a test suite *S* and an ordering *Ord*$_S$ for test cases in this suite:

$$cumCCov(t_i) = \bigcup_{j=1}^{i} CCov(t_j)$$

where t$_i$ is a test case in the suite. The cumulative code coverage for t_i is computed by summing single code coverage (i.e., the code covered only by the test case) of all those test cases from t_0 to t_{i-1}.

Requirements. The capability of a test case in exercising users' and/or application requirements depends on: *(i)* the amount of requirements covered by this test case and *(ii)* the relevance of covered requirements. Similarly to code coverage measure, we defined and used *RCov(t)* and its weighted variant *WRCov(t)*. In particular, *RCov(t)* is the measure of requirements coverage for test case *t* and measures application requirements exercised during the execution of *t*. On the other hand, *WRCov(t)* measures the coverage for a test case as follows:

$$WRCov(t) = \sum_{r \in Requirements} \begin{cases} w_r & r \in ReqsCovered \\ 0 & otherwise \end{cases}$$

Requirements is a set containing the requirements of application under test. *ReqsCovered* is the set of requirements covered by the execution of test case *t*, obtained by means of traceability links recovered by applying our approach. On the other hand, *r* is one of application requirements and w_r ($0 \leq w_r \leq 1$) is weight associated to this requirement. Requirements weight w_r can be defined in several ways according to fault-proneness of application requirements. The larger w_r, the greater the fault-proneness of requirement is. Our metric-based requirements prioritization technique automatically identifies fault-prone application requirements, thus to be highly weighted when computing the coverage.

Given a test suite *S* and a possible ordering *Ord*$_S$ for test cases of this suite, we define:

$$cumRCov(t_i) = \bigcup_{j=1}^{i} RCov(t_j)$$

where t$_i$ is a test case of the suite. Cumulative requirements coverage for test case t_i is computed by summing single requirements coverage (i.e., the requirements covered only by the test case) of all those test cases from t_0 to t_{i-1}.

Execution cost. The execution cost of a test case can be approximated by the time required to its execution. If the implementation of test cases is available, their execution can be profiled to collect information about running time. We defined *Cost(t)* as the time required to execute test case *t*.

Given a test suite *S* and an ordering *Ord*$_S$ for test cases of this suite, we defined *cumCost(t$_i$)*, where t$_i$ is one of the test case of the suite. It represents the cumulative execution of test case t_i and it is computed as the sum of execution costs of test cases preceding test case $t_i \in$ *Ord*$_S$.

Cost(suite) is the overall cost of test cases and is computed as the sum of execution costs of all the test cases. We then define:

$$InverseCost(t_i) = Cost(suite) - \sum_{j=1}^{i} Cost(t_j)$$

5.2 Automatic Weighting

Our metric-based approach automatically weights both code w_s and requirements w_r of the application under test. In particular, we apply code metrics to measure a Maintainability Index for each Java class (MI_{class}). This index estimates the fault-proneness of each class. We use such an estimation for defining an order of the application classes. To prioritize all the requirements according to how they are implemented, we also compute a Maintainability Index for each of these requirement (MI_{req}). The idea

TABLE 1
Metrics used for the automatic weighting

Metric	Ref.	Property	Definition
Class-level Metrics			
(CBO) Coupling Between Objects	[42]	Coupling	It is the number of classes to which a class is coupled
(RFC) Response For a Class	[42]	Coupling	It is the set of methods that can potentially be executed in response to a message received by an object of the class
($LCOM$) Lack Of Cohesion on Methods	[42]	Cohesion	It describes the lack of cohesion among methods of a class
($LOCs$) Lines of Code	-	Size	It counts the lines of code of a class
(NOM) Number of methods	-	Size	It counts the number of methods of a class
(DIT) Depth of Inheritance Tree	[42]	Inheritance	It is the length of the class from the root of the inheritance tree
(NOC) Number of Children	[42]	Complexity	It is the number of immediate subclasses of the class in the class hierarchy
(MCC) McCabe Cyclomatic Complexity	[42]	Complexity	It is (median of) the number of flows thought the code of the method of a class
(WMC) Weighted Methods per Class	[42]	Complexity	It is the sum of the MCC for all methods in a class
Requirements-level Metrics			
(NC) Number of Classes	[43]	Size	It is the number of classes implementing a requirements
(CDC) Requirements diffusion over components	[43]	Scattering	It is the number of classes that contribute to the implementation of the target requirements, among those of the application
(CDC+) CDC with similarity	-	Scattering	It is a variant of CDC in which the contribution of each class is weighted according to the similarity of each class with the requirements definition
(ShR) Shared among Requirements	[44]	Tangling	It expresses the degree of classes that implement a requirements and that are shared with, at least, another requirements of the application
(ShR+) ShR with similarity	-	Tangling	It is a variant of ShR in which the contribution of each class is weighted according to the similarity of each class with the definition of the requirements under analysis
(IN) Contained Requirements	[44]	Inheritance	It is the number of requirements whose implementation is entirely contained in the target requirements

that guides both code and requirements prioritization is to realize a most critical first strategy. That is, we aim at increasing the possibility of testing the most critical classes and requirements before the other classes and requirements.

Our automatic weighting approach is composed of the following steps:

1) **Recovering traceability links**. Links among software artifacts (i.e., source code and requirements) are recovered by applying LSI;
2) **Computing metrics**. For each class, we measure a set of metrics such as: size, complexity, coupling, and cohesion. For each requirement, a set of metrics is also computed to measure properties characterizing requirements: size, complexity, coupling, cohesion, scattering, and tangling degree.
3) **Estimating maintainability indexes**. The computed metrics are used in a software quality model to compute the maintainability index for each class and each requirement based on their actual implementation in the source code. Classes and requirements are ordered by ranking according to their maintainability index.

5.2.1 Computing metrics

In Table 1, we summarize the used metrics. This table shows the following information: the name of each metric, the reference to the paper that originally defined it, the measured software property, and an intuitive definition of that metric. A formal and precise definition of such metrics is beyond the scope of our paper.

We adopt the Class-level Metrics (Table 1 top) to compute MI_{class}. On the other hand, to compute MI_{req}, we adopt two distinctive sets of metrics working at the following two levels of granularity:

1) traditional object-oriented size metrics working at *class-level* (Table 1 top), i.e., for each class of the target application we measure each metric;
2) concern-oriented metrics[3] working at *requirements-level* (Table 1 bottom), i.e., for each requirement we measure such metrics inspired to the concern ones [45].

The rationale behind the use of these two types of metrics is that class-level metrics measure the classes composing each requirement in isolation, while requirements-level metrics let us relate the requirements implemented with the one of the other requirements of the application.

5.2.2 Estimating maintainability indexes

The maintainability index is obtained by means of the following three steps:

Outliers identification. After computing the code and requirements-level metrics, we identify *outliers* [47]. For a given metric, outliers are elements (i.e., classes and requirements) whose values for such a metric exceed a given threshold that is obtained on the base of the values the other elements have for that metric. In our case, the outliers are those elements having metric values within the highest/lowest 15% of the value range defined by all elements of the application [47]. For instance, if the CBO value ranges between 0 and 56. Given two classes having $CBO(c1) = 52$ and $CBO(c2) = 35$, then *c1* is an outliers for CBO (i.e., the value of *c1* is in the range 85-100% of CBO), while *c2* is not an outliers.

Software Quality model. In Table 2, we present the software quality models (at class- and requirements-level) used to compute the maintainability index MI for each class c and requirements r, starting from the two sets of

3. Their common goal regards the association of concern property quantification with the impact on modularity flaws [45]. A concern is any consideration that might impact the implementation of a program, whilst concern measures lead to a shift in the measurement process instead of quantifying properties of a particular module. These measures quantify properties of one or multiple concerns with respect to the underlying modular structure [46].

TABLE 2
Software Quality Model

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

metrics in Table 1. $MI(c)$ represents an estimation of the maintainability degree of each application code class c by considering its structural properties [47]. $MI(r)$ represents an estimation of the maintainability degree of the software code implementing the requirements r by considering its structural properties. The used models are inspired by those by Lincke *et al.* [47]. In these models, the metrics are weighted according to the software property they measure. As for class-level and consistently with Lincke, we chose: 2 for coupling, cohesion, and inheritance metrics, while 1 for the other metrics. As for requirements-level, we chose: 2 for size and scattering, while 1 for the other metrics.

By knowing the outliers and using a software quality model such as the one in Table 2, we can compute a maintainability index for each application element by aggregating the metrics in Table 1 according to their weights in the used models as follows:

$$MI(element) = \frac{\sum_{m \in Metrics}(W_m * O_m)}{\sum(W_m)}$$

$Metrics$ is the set of metrics, W_m is the weight of the metrics m in the model and O_m is the number of outliers elements for the metrics in the considered model. The $MI(element)$ value ranges in between 0 and 1. The value 0 is the best possible index (i.e., no outliers elements), while 1 is the worst (i.e., all elements are outliers). For instance, if a class *c1* is an outliers only for CBO, LOCs, and DIT, then, referring to Table 2, $MI(c1) = 5/13 = 0.385$, so, *c1* has a maintainability index of 0.385, i.e., 38.5%.

Maintainability index computation. To compute the maintainability index $MI_{class}(c)$, we use the class-level quality model (Table 2 top). Then, we compute a maintainability index $MI_{req}(r)$ for each requirement r of the target application. To this aim, two additional maintainability indexes are computed: $MI_C(r)$ and $MI_R(r)$. $MI_C(r)$ is computed by averaging the maintainability index MI_{class} of the classes that implement r (on the base of traceability links), while $MI_R(r)$ is computed by applying the requirements-level quality model for r (i.e., using the requirements-level metrics in Table 1). By averaging $MI_C(r)$ and $MI_R(r)$ for each requirement r, we estimate the maintainability for r by considering the properties, at the same time, of the code implementing r in isolation and the implementation of r with respect to the other requirements [48].

Application source code classes and requirements are ordered by ranking the classes and the requirements according to their estimated maintainability index (MI_{class} and MI_{req}, respectively).

TABLE 3
Example: statements, classes, requirements, and test case definition

S=$\{St, C, R\}$ St=$\langle s1, s2, s3, s4, s5\rangle$ C=$\langle c1, c2, c3, c4\rangle$ R=$\langle r1, r2, r3\rangle$		
Statements	**Class**	**Reqs**
s1	c1	r2
s2	c2	r3
s3	c2	r1
s4	c3	r3
s5	c4	r3

Test Case	Cost (seconds)	Statements	Reqs.
t1	20	s1, s3	r1, r2
t2	100	s3, s4, s5	r1, r3
t3	50	s1, s2, s3	r1, r2, r3

MI$_{class}$		**MI**$_{req}$	
MI$_{class}$(c1)=0.02	MI$_{class}$(c2)=0.3	MI$_{req}$(r1)=0.5	MI$_{req}$(r2)=0.2
MI$_{class}$(c3)=0.2	MI$_{class}$(c4)=0.8	MI$_{req}$(r3)=0.75	

5.3 Identifying Cumulative Test Orderings

For each test case t_i of a given test ordering *Ord*$_S$, the measures *cumCCov(t*$_i$*)*, *cumRCov(t*$_i$*)* and *InverseCost(t*$_i$*)* are computed considering the position of t_i in *Ord*$_S$. Then, we computed the area of the curves obtained by plotting the values of the metric (on X axes) with respect to the test cases in *Ord*$_S$ (Y axes) in a *Cartesian* plan. To get a numerical approximation of that area, we used the *Trapezoidal* rule [49]. It computes the area of a curve as the area of a linear function that approximates that curve.

For *Ord*$_S$ and each cumulative measure, the area (Area Under the Curve, *AUC*, from here on) estimates the code coverage *AUCcode(Ord*$_S$*)*, the requirements coverage *AUCreq(Ord*$_S$*)*, and the execution cost *AUCcost(Ord*$_S$*)*, respectively. The area indicates how fast the test ordering *Ord*$_S$ converges. The larger *AUC*, the faster this test case ordering converges.

5.4 Example

As an example, let us consider a system that implements three requirements (r1-r3) and that is composed of four classes (c1-c4) and five statements (s1-s5). Table 3 (top) details relationship among statements, classes and requirements. For instance, statement s1 is part of class c1 and it contributes to realize r2. Table 3 (middle) shows a test suite composed of three test cases (*t1, t2, t3*) and it shows cost and coverage information for each test case as well. For instance, test case t1 costs 20 (seconds) and it covers s1 and s2, and it tests requirements r1 and r2. We assume that coverage information contained in Table 3 has been achieved by applying our IR-based traceability recovery approach. Table 3 (bottom) shows possible values of maintainability index MI computed for each class and each requirement of system under test and used in test suite prioritization to weight metrics for measuring code and requirements coverage of each test case. We assume that such values have been automatically computed. Considering two possible test orderings for test cases shown in Table 3, namely *Ord1*$_S$=$\langle t1, t2, t3\rangle$ and *Ord2*$_S$=$\langle t3, t1, t2\rangle$. Values of *AUC* for the two test case orderings are shown in Table 4. This table also reports some details for the computation of each measure adopted to compute AUC values. In particular, the table shows measures computed for two test orderings by non-weighting (column $w_x = 1$) measures and by automatically weighting (column $w_x = MI$) such measures with maintainability index MI. For example, we can see in Table 4 that cumCCov(t1) corresponds to 2 in non-weighted measures (where each weight $w_x = 1$) for test ordering *Ord1*$_S$. On the other hand, cumCCov(t1) corresponds to 0.32 (MI_{class}(c1)$|\{$s1$\}|$+MI_{class}(c2)$|\{$s3$\}|$=0.02*1+0.3*1) in MI-based weighted measures (where each weight $w_x = MI_{class}(x)$). By comparing *Ord1*$_S$ and *Ord2*$_S$ according to the three *AUC* measures computed using both non-weighted and weighted approaches, we can note that *Ord1*$_S$ and *Ord2*$_S$ have the same AUCcode, but they are different in terms of AUCreq and AUCcost, namely *Ord2*$_S$ has a higher AUC in both AUCreq and AUCcost. In addition, we can note that *Ord1*$_S$ has a higher AUCcode than *Ord2*$_S$ while this former still preserve a higher value for both AUCreq and AUCcost. Hence, we can deduce that: *(i)* if we consider non-weighted approach, *Ord2*$_S$ outperforms *Ord1*$_S$ having greater or, at least, not inferior values in all three measures and *(ii)* if we consider weighted approach, no one test ordering outperforms other one, in all three considered dimensions. This example suggests that our automatic weighting lets us to work actually at a fine-grained granularity and gives more relevance to key portions of source code and requirements.

TABLE 4
Example of AUC measures and comparison between different test cases orderings (*Ord1*$_S$ and *Ord2*$_S$)

Ord1$_S$=$\langle t1, t2, t3\rangle$

Measure	Computation	w_x	
		1	MI_x
cumCCov(t1)	$w_x|\{$s1$\}|$+ $w_x|\{$s3$\}|$	2	0.32
cumCCov(t2)	cumCCov(t1)+$w_x|\{$s4$\}|$ + $w_x|\{$s5$\}|$	4	1.32
cumCCov(t3)	cumCCov(t2) +$w_x|\{$s2$\}|$	5	1.62
AUCcode		***8.5***	***2.45***
cumRCov(t1)	$w_x|\{$r1$\}|$+ $w_x|\{$r2$\}|$	2	0.7
cumRCov(t2)	cumRCov(t1) +$w_x|\{$r3$\}|$	3	1.45
cumRCov(t3)	cumRCov(t2)	3	1.45
AUCreq		*6.5*	*2.875*
Cost(suite)	170		
InvCost(t1)	150		
InvCost(t2)	50		
InvCost(t3)	0		
AUCcost	*200*		

Ord2$_S$=$\langle t3, t1, t2\rangle$

Measure	Computation	w_x	
		1	MI_x
cumCCov(t3)	$w_x|\{$s1$\}|$+$w_x|\{$s2$\}|$+$w_x|\{$s3$\}|$	3	0.62
cumCCov(t1)	cumCCov(t3)	3	0.62
cumCCov(t2)	cumCCov(t1)+$w_x|\{$s4$\}|$ + $w_x|\{$s5$\}|$	5	1.62
AUCcode		***8.5***	*2.05*
cumRCov(t3)	$w_x|\{$r1$\}|$+$w_x|\{$r2$\}|$+$w_x|\{$r3$\}|$	3	1.45
cumRCov(t1)	cumRCov(t3)	3	1.45
cumRCov(t2)	cumRCov(t1)	3	1.45
AUCreq		***7.5***	***3.625***
Cost(suite)	170		
InvCost(t3)	120		
InvCost(t1)	100		
InvCost(t2)	0		
AUCcost	***220***		

6 Multi-Objective Prioritization

NSGA-II uses a set of genetic operators (i.e., crossover, mutation, and selection) to iteratively evolve an initial population of candidate solutions. In our case, candidate solutions are test cases orderings. Evolution is guided by an objective function (i.e., the fitness function) that evaluates each candidate solution along considered dimensions. In each iteration, the *Pareto* front of best alternative solutions is generated from evolved population. The front contains the set of non-dominated solutions, i.e., those solutions that are not inferior (dominated) to any other solution in *all* considered dimensions. Population evolution is iterated until a (predefined) maximum number of iterations is reached.

In our case, a Pareto front represents the optimal trade-off between the three dimensions determined by NSGA-II. The tester can then inspect a Pareto front to find the best compromise between having a test case ordering that balances code coverage, requirements coverage, and execution cost or alternatively having a test case ordering that maximizes one/two dimension/s penalizing the remaining one/s.

Our proposed process can be summarized as follows:

1) **Solution Encoding.** A solution is a possible ordering of the test cases under analysis. *Ord*$_S$ represents an execution order for the test cases of suite *S*. The solution space for the test case prioritization problem is the set of permutations of test case orderings. A test case ordering is represented as an ordered sequence of integers, where each integer represents the identifier of a test case.
2) **Initialization.** The starting population is initialized randomly selecting a sub-set of possible test case orderings among all possible permutations of test cases (i.e., the solution space).
3) **Genetic Operators.** For the evolution of permutation-based encoding for the solutions, we exploited standard operators as described in [50]. As mutation operator, we used *SWAP-Mutation* in which two randomly chosen permutation elements of the solution are swapped. The adopted crossover operator is *PMX-Crossover* in which a pair of solutions is recombined by randomly choosing an intermediate point and swapping permutation elements at that point among both solutions. Finally, we used *binary tournament* as selection operator. Two solutions are randomly chosen and the fitter of the two is the one that survives in the next population.
4) **Fitness Functions.** Since our goal is to maximize the three considered dimensions, each candidate solution in the population is evaluated by our objective function based on: *AUCcode(Ord*$_S$*)*, *AUCreq(Ord*$_S$*)*, and *AUCcost(Ord*$_S$*)*. The larger these values, the faster

the convergence of a test case ordering is.

7 Experimental Assessment

To evaluate of our approach, we have developed a prototype of a supporting system. It integrates and extends the following two tool prototypes: *(i) MOTCP* [51] that implements our previous proposed prioritization technique [4] and represents the base for our new approach and *(ii)* SWT-Metrics [48], which implements our automatic weighting approach to prioritize software artifacts. *MOTCP+* is the name of the tool implementing our new prioritize technique. It is composed of a number of software components having the purpose of preparing data related to metrics and traceability links and executing the prioritization process as shown in Figure 1. In particular, there is a component in charge of recovering traceability links among software artifacts. It integrates and extends Traceclipse [30]. We implemented a component to compute class-level metrics and requirements-level metrics and a component to compute the maintainability index of each code class and requirement and to determine their orderings. Finally, our three-objective test case prioritization algorithm was implemented in a component that integrated the implementation of NSGA-II available in JMetal[4] meta-heuristics library [52].

According to the Goal Question Metrics (GQM) template by Basili and Rombach [53], the goal of our experiment can be summarized as follows:

1) **Analyze** the use of our proposal **for the purpose** of evaluating its support in the prioritization of test cases **with respect to** effectiveness, sensitivity, and robustness **from the point of view** of the researcher **in the context of** Java applications and **from the point of view** of the practitioner assessing whether our proposal is a viable solution **in the context of** his/her own company.

The GQM formalism ensures that important aspects are defined before planning and execution of our experiment took place [54].

According to our experiment goal, we compare our proposal with traditional test cases prioritization techniques [12], [13], [14], namely random prioritization (*Rand*), code coverage (*CodeCov*), and additional code coverage (*AddCodeCov*) prioritization. Similarly to Yoo and Harman [18], we also applied NSGA-II on the dimensions: code coverage and execution cost of test cases. This represents another baseline for comparison (*NSGAIIdim2*). An additional baseline for comparison is our previous multi-objective technique [4]. It used code coverage, requirements coverage, and execution time of test cases without applying the automatic weighting scheme we present in this paper.

The comparison has been performed with respect to the following criteria:

- **Effectiveness.** It concerns the capability of test case orderings in revealing faults.
- **Sensitivity**. From a tester's perspective, this criterion provides an indication on the capability of test case orderings in revealing faults with a high severity and relevance with respect to application requirements.

Only in the case of our proposal, we also analyzed its **Robustness** with respect to the goodness of recovered traceability links. Robustness gives us an idea about the capability of our test case prioritization approach of adequately working in presence of incomplete or spurious/wrong traceability links.

To have a deeper understanding of results, we also perform: *(i)* an analysis of the generated **Pareto Fronts** and of **impact of each metric** used by our tool to find optimal solutions and *(ii)* an analysis of possible **co-factors** characterizing experimental objects and artifacts as well as relationships among them. Among analyzed co-factors, we consider: the size of applications and their test suites, the number of requirements, the relationships between test cases and requirements, the capability of test suites in revealing faults, the test case redundancy, and the distribution of faults in source code and requirements.

7.1 Evaluation Measures

The Average Percentage of Fault Detected (*APFD*) is the measure conventionally adopted to evaluate test case orderings [13]. Given a test suite *S* containing *n* test cases and let *F* be the set of *m* faults revealed by *S*. For an ordering *S'* of *S*, let SF_i be the position of first test case $s \in S'$ that reveals the *i-th* fault. The APFD value for *S'* is computed as follows:

$$APFD = 1 - \frac{SF_1 + ... + SF_m}{nm} + \frac{1}{2n}$$

We run an approach (e.g., AddCodeCov or MOTCP+) on a given application, thus obtaining an ordering *S'*. A number of versions of that application are obtained seeding one fault per time in its source code [13]. That is, each version contains only one injected fault. To assess the capability of *S'* in detecting faults the APFD value is computed with respect to the obtained versions of the original application. A high APFD value signifies a fast fault-detection rate of the ordering *S'*.

The APFD-based test case prioritization evaluation assumes that test costs and fault severity and relevance are all uniform [55]. However, test costs and fault severity can vary widely in a real-life context. Hence, to get a quantitative measure of **Effectiveness** and **Sensitivity**, we consider three variants of that measure: $APFD_{all}$ computed checking all injected faults, $APFD_{ftype1}$ computed checking the subset of severe faults; and $APFD_{ftype2}$ computed checking the subset of faults related to relevant requirements. In particular, Effectiveness is estimated by $APFD_{all}$, while APFD_{ftype1} and APFD_{ftype2} estimate Sensitivity.

We statistically analyze results achieved by MOTCP+ and baseline techniques. The tested null hypothesis is:

4. http://jmetal.sourceforge.net/

TABLE 5
Objects under study

Application	Size (LOCs)	Test Cases	Req.s	Faults	Web site
LaTazza	2k	33	10	12	-
AveCalc	2k	47	10	15	-
CommonsProxy	5k	179	10	10	http://commons.apache.org/proxy
DBUtils	5k	225	12	14	http://commons.apache.org/dbutils
iTrust	15k	919	15	21	http://agile.csc.ncsu.edu/iTrust
CommonsCodec	17k	608	19	20	http://commons.apache.org/codec
JTidy	20k	289	25	15	http://jtidy.sourceforge.net
Woden	22k	263	24	19	http://ws.apache.org/woden
Log4J	25k	1029	24	20	http://logging.apache.org/log4j
Betwixt	25k	325	18	20	http://commons.apache.org/dormant/commons-betwixt
JXPath	25k	386	20	20	http://commons.apache.org/jxpath
CommonsIO	25k	859	18	20	http://commons.apache.org/
CommonsBcel	30k	75	20	20	http://commons.apache.org/bcel
CommonsBeanUtils	32k	1556	26	22	http://commons.apache.org/beanutils
XMLGraphics	34k	196	24	15	http://xmlgraphics.apache.org
XMLSecurity	40k	92	23	15	http://santuario.apache.org
CommonsCollections	50k	798	17	20	http://commons.apache.org/collections
Pmd	55k	698	20	20	http://pmd.sourceforge.net
CommonsLang	60k	2307	16	20	http://commons.apache.org/lang
Jabref	70k	213	31	20	http://jabref.sourceforge.net
Xerces	138k	376	20	20	http://xerces.apache.org

NH_{faults} - *there is no difference in the APFD values obtained on the suites generated by applying the different approaches.*

To test this null hypothesis, we conducted pairwise comparisons among results achieved for test suites generated by the techniques by using the non-parametric one-tailed Mann-Whitney test since we expect that our novel approach will obtain better results than other techniques. We use the Benjamini-Hochberg [56] correction for the compensation of repeated statistical tests.

To analyze **Pareto Front's metrics**, we adopt the Spearman's Rank correlation coefficient (ρ) to estimate collinearity, if any, of the three metrics used to build Pareto fronts. The Spearman's Rank correlation coefficient measures correlation between a pair of variables. The returned value ranges in between -1 and +1, where +1 indicates perfect correlation and -1 indicates a perfect inverse correlation. We also apply Principal Components Analysis (PCA) to check whether metrics used to built each front are correlated each other, thus discovering those metrics that are dominant and those redundant. PCA is a non-parametric statistical technique that estimates interrelation degree of variables for identifying underlying structures, if any, and combining variables into smaller sets of linearly uncorrelated variables, called *principal components* (PCs). By applying PCA, we aim at checking the presence of interrelations among metrics on test suites composing Pareto fronts. The defined null hypothesis is:

NH_{pareto} - *no correlations exist among our Pareto's metrics.*

To evaluate the impact of possible **co-factors** on achieved results, we mainly applied a two-way permutation test [57]. Our null hypothesis is:

$NH_{co-factors}$ - *there is no significant impact of the considered co-factor/s on APFD values.*

In all performed statistical tests, we decided (as customary) to accept a probability of 5% of committing a Type-1-Error [54], namely a null hypothesis is rejected if the p-value returned by a statistical test is less than 0.05.

7.2 Experimental Objects

We considered 21 Java applications form different application domains as experimental objects. These applications range from small to large in terms of size and implemented functionality. Table 5 summarizes the size of each application in terms of lines of code, as well as the number of test cases, requirements, and faults, and shows links to application websites. The considered 21 Java applications were chosen primarily because of the availability of software artifacts we needed to apply our technique (e.g., textual description of the application requirements).

7.3 Procedure

For each application, we applied the following experimental procedure:

1) Collecting available artifacts. For each application, we collected requirements specifications, source code, and JUnit test cases.
2) Recovering the traceability links. We used the following set-up: *k*=300; *constant threshold*=0.1.
3) Applying MOTCP+, MOTCP, Rand, CodCov, AddCodeCov, and NSGAIIdim2. We ran MOTCP, MOTCP+, and NSGAIIdim2 with the following set-up: *population size*=2*'test suite size'; *max iterations*=1000; *crossover probability*=0.9; *mutation probability*=1/'test suite size'. Since Rand has a non-deterministic behavior, we ran it several times (i.e., 30 times) and then we evaluated all generated solutions. We report descriptive statistics on the values of obtained solutions (min, $median$, $mean$, and max). On the other hand, since MOTCP, MOTCP+, and

NSGAIIdim2 are expected to generate sets of equivalent good solutions per execution (Pareto front), we evaluated all solutions in an obtained front by considering the following descriptive statistics on used measures: min, $median$, $mean$, and max. This allowed us to better analyze the behavior of studied techniques. It easily follows that the meaning of min, $median$, $mean$, and max is different between Rand and MOTCP, MOTCP+, and NSGAIIdim2.

4) Injecting m faults of different severity (i.e., high and low) in the source code of a given application, thus producing m buggy versions. For example, m is equal to 12 for LaTazza (see Table 5), meaning that we obtained 12 different faulty versions for that application. The injection of faults is critical because hand-seeded faults may not be representative of real faults [58]. Therefore, we applied a repeatable process to inject faults that are similar as much as possible to actual ones [59]. This process is based on the following steps that are executed sequentially:

    a) Analyzing online bug tracker of a given application to find documented failures.
    b) Faults generated these documented failures are selected according to the following criteria: *(i)* status of the fault is closed/solved; *(ii)* deviation between observed and specified behaviors is clearly described from a functional point of view; and *(iii)* it is possible to link the fault to the code where this fault was present. To get this link, we needed the following information: application version where a failure was observed; code (i.e., class name at least) containing that fault; version of application where that fault was fixed and how it was fixed.
    c) Analyzing code patches posted to fix a fault (if any) and code affected by this fault. The goal here is to recover information on how to reproduce the original failure.
    d) Replicating each failure by injecting a fault in the source code of the original application. As mentioned before, one fault per time is injected in that code, so obtaining one faulty version for each injected fault . We check the capability of a test suite in revealing the failure associated to each injected fault. If at least one test case fails (i.e., the failure is detected by our test suite), we add the injected fault to the set of faults of our experimental investigation.
    e) Examining each injected fault to understand if it can be considered *severe* and/or *related to relevant requirements*. To classify a fault as severe, we primary consider its severity and priority fields in bug report. For example, a fault is severe if both its priority and its severity are high. To classify faults with respect to relevant requirements, we considered which application functionality is affected by fault and relevance of that functionality from the user's perspective. For example, a fault is considered related to a relevant requirement if application documentation lists this requirement as one of most important ones.

   To reduce as much as possible threats related to representativeness of hand-seeded faults, the injection process was performed by an author involved neither in the definition of our prioritization technique nor in the execution of experiment. In the Appendix A , we show an example of application of the described injection process.

5) Executing test case orderings. Test case orderings obtained by the techniques is executed for testing each version of the application.
6) Computing $APFD_{all}$, $APFD_{ftype1}$ and $APFD_{ftype2}$ for each studied test prioritization technique, application, and ordering.
7) Executing steps from 2 to 7. In the second iteration, we randomly changed and/or removed 10% of the recovered traceability links (step 2) to estimate the Robustness of the approach.
8) Analyzing collected data. Our analysis procedure is applied on these data.

7.3.1 *Threats to validity*

Used experimental objects (i.e., applications) and artifacts (i.e., source code, test cases, and requirements) might threaten the validity of our results. To deal with these threats, we used a large set of applications having different characteristics (e.g., from small to large) and application domains (e.g., bibliography reference manager). As far as application artifacts are concerned, we exploited as much as possible those provided by original developers. If not available, we reconstructed them by looking at the documentation provided by the developers in user manuals and APIs.

Another threat to validity is the set of injected faults, as well as their distribution in application code. We are conscious that different sets of faults could lead to different results. With the aim of limiting such a threat, we used an experimental process [59] that exploits actual faults described into application bug tracker systems. Therefore, we analyzed the bug tracker of each application and selected not trivial and critical faults that we were able to reproduce. Since it is rare to have an application with many faults, we produced a version for each fault injected. As for test case execution cost, we only considered the time needed to execute test cases. This choice represents a limitation for the applicability of our approach [60], [61]. However, our approach can be easily extended to take also into account additional related costs for regression testing (e.g., the time to inspect the results). This is the subject of future work.

The set up of the experiment represents another threat to the validity of results. In particular, the number of runs for Rand and parameters chosen for the recovery of links among software artifacts and the parameters chosen in the multi-objective algorithm could potentially affect results.

TABLE 6
APFD of AveCalc, LaTazza, DBUtils, CommonsProxy and iTrust

	APFD														
	AveCalc			LaTazza			DBUtils			CommonsProxy			iTrust		
	all	$ftype1$	$ftype2$	all	$ftype1$	$ftype2$	all	$ftype1$	$ftype2$	all	$ftype1$	$ftype2$	all	$ftype1$	$ftype2$
Traceability Links															
Rand$_{min}$	0.59	0.74	0.62	0.66	0.54	0.58	0.41	0.41	0.27	0.37	0.34	0.17	0.52	0.49	0.47
Rand$_{mean}$	0.73	0.82	0.84	0.75	0.74	0.83	0.52	0.57	0.47	0.53	0.55	0.5	0.63	0.65	0.63
Rand$_{median}$	0.74	0.85	0.8	0.75	0.75	0.72	0.51	0.57	0.46	0.53	0.56	0.5	0.63	0.66	0.66
Rand$_{max}$	0.83	0.93	0.91	0.81	0.88	0.88	0.65	0.8	0.75	0.65	0.77	0.74	0.73	0.81	0.79
CodeCov	0.77	0.78	0.81	0.6	0.72	0.56	0.4	0.34	0.47	0.56	0.62	0.45	0.56	0.6	0.58
AddCodeCov	0.79	0.8	0.92	0.6	0.72	0.56	0.55	0.71	0.4	0.59	0.65	0.47	0.7	0.71	0.69
NSGAIIdim2$_{min}$	0.72	0.79	0.75	0.78	0.7	0.83	0.46	0.43	0.28	0.39	0.27	0.35	0.52	0.51	0.51
NSGAIIdim2$_{mean}$	0.78	0.82	0.84	0.78	0.7	0.83	0.52	0.56	0.46	0.48	0.45	0.46	0.6	0.62	0.6
NSGAIIdim2$_{median}$	0.79	0.83	0.86	0.78	0.7	0.83	0.52	0.53	0.52	0.47	0.41	0.47	0.61	0.64	0.61
NSGAIIdim2$_{max}$	0.87	0.91	0.94	0.78	0.7	0.83	0.61	0.75	0.54	0.59	0.75	0.57	0.63	0.67	0.64
MOTCP$_{min}$	0.72	0.75	0.76	0.75	0.65	0.72	0.46	0.62	0.25	0.42	0.35	0.38	0.49	0.4	0.49
MOTCP$_{mean}$	0.78	0.84	0.83	0.77	0.7	0.75	0.53	0.71	0.43	0.53	0.47	0.53	0.63	0.64	0.65
MOTCP$_{median}$	0.79	0.84	0.83	0.77	0.7	0.75	0.51	0.71	0.44	0.52	0.49	0.5	0.63	0.63	0.63
MOTCP$_{max}$	0.86	0.92	0.94	0.79	0.75	0.79	0.65	0.9	0.52	0.65	0.77	0.8	0.72	0.73	0.7
MOTCP+$_{min}$	0.62	0.69	0.69	0.77	0.77	0.76	0.45	0.42	0.31	0.31	0.27	0.21	0.52	0.6	0.52
MOTCP+$_{mean}$	0.79	0.83	0.87	0.82	0.82	0.88	0.57	0.7	0.47	0.47	0.41	0.45	0.61	0.67	0.61
MOTCP+$_{median}$	0.8	0.82	0.88	0.82	0.84	0.91	0.56	0.7	0.48	0.47	0.46	0.46	0.61	0.66	0.61
MOTCP+$_{max}$	0.9	0.94	0.94	0.85	0.86	0.93	0.67	0.84	0.63	0.67	0.69	0.72	0.8	0.81	0.81
Incomplete Traceability Links															
MOTCP+$_{min}$	0.69	0.72	0.82	0.71	0.75	0.76	0.44	0.44	0.35	0.3	0.25	0.19	0.53	0.53	0.52
MOTCP+$_{mean}$	0.79	0.81	0.88	0.75	0.8	0.79	0.56	0.59	0.52	0.5	0.5	0.49	0.6	0.65	0.62
MOTCP+$_{median}$	0.79	0.8	0.89	0.75	0.79	0.79	0.56	0.57	0.53	0.5	0.5	0.53	0.59	0.63	0.61
MOTCP+$_{max}$	0.88	0.92	0.92	0.78	0.87	0.82	0.62	0.77	0.59	0.65	0.76	0.59	0.69	0.76	0.71

TABLE 7
APFD of JTidy, CommonsCodec, Woden, Log4J and Betwixt

	APFD														
	CommonsCodec			JTidy			Woden			Log4J			Betwixt		
	all	$ftype1$	$ftype2$	all	$ftype1$	$ftype2$	all	$ftype1$	$ftype2$	all	$ftype1$	$ftype2$	all	$ftype1$	$ftype2$
Traceability Links															
Rand$_{min}$	0.36	0.31	0.35	0.43	0.19	0.19	0.44	0.38	0.25	0.55	0.4	0.51	0.46	0.5	0.29
Rand$_{mean}$	0.57	0.57	0.6	0.57	0.51	0.54	0.56	0.55	0.53	0.63	0.61	0.67	0.58	0.63	0.57
Rand$_{median}$	0.58	0.6	0.62	0.57	0.47	0.47	0.57	0.55	0.51	0.63	0.64	0.66	0.58	0.63	0.59
Rand$_{max}$	0.66	0.75	0.73	0.65	0.73	0.73	0.65	0.71	0.71	0.75	0.78	0.88	0.69	0.8	0.77
CodeCov	0.58	0.46	0.4	0.46	0.27	0.27	0.5	0.53	0.57	0.48	0.58	0.7	0.52	0.66	0.5
AddCodeCov	0.53	0.43	0.44	0.5	0.24	0.24	0.54	0.44	0.58	0.76	0.69	0.84	0.58	0.66	0.52
NSGAIIdim2$_{min}$	0.49	0.45	0.45	0.63	0.6	0.55	0.39	0.27	0.18	0.61	0.52	0.53	0.48	0.49	0.43
NSGAIIdim2$_{mean}$	0.59	0.6	0.58	0.71	0.68	0.62	0.42	0.39	0.34	0.7	0.67	0.65	0.56	0.69	0.54
NSGAIIdim2$_{median}$	0.59	0.64	0.59	0.73	0.71	0.57	0.42	0.41	0.39	0.71	0.66	0.64	0.56	0.71	0.52
NSGAIIdim2$_{max}$	0.66	0.71	0.69	0.8	0.75	0.81	0.44	0.46	0.4	0.78	0.78	0.83	0.68	0.82	0.77
MOTCP$_{min}$	0.42	0.32	0.43	0.58	0.46	0.46	0.43	0.39	0.35	0.5	0.45	0.54	0.48	0.55	0.4
MOTCP$_{mean}$	0.56	0.55	0.57	0.64	0.56	0.71	0.49	0.48	0.48	0.65	0.59	0.72	0.6	0.7	0.58
MOTCP$_{median}$	0.56	0.55	0.56	0.61	0.59	0.59	0.5	0.47	0.48	0.65	0.59	0.71	0.61	0.71	0.59
MOTCP$_{max}$	0.67	0.77	0.73	0.71	0.68	0.68	0.57	0.61	0.64	0.75	0.7	0.92	0.67	0.86	0.68
MOTCP+$_{min}$	0.45	0.35	0.37	0.47	0.43	0.43	0.43	0.32	0.27	0.54	0.54	0.49	0.5	0.49	0.47
MOTCP+$_{mean}$	0.54	0.5	0.54	0.62	0.58	0.58	0.54	0.5	0.53	0.67	0.66	0.72	0.61	0.67	0.67
MOTCP+$_{median}$	0.54	0.48	0.51	0.62	0.59	0.59	0.53	0.48	0.54	0.66	0.67	0.71	0.61	0.66	0.61
MOTCP+$_{max}$	0.69	0.78	0.79	0.77	0.85	0.85	0.68	0.76	0.74	0.76	0.82	0.87	0.74	0.86	0.83
Incomplete Traceability Links															
MOTCP+$_{min}$	0.45	0.36	0.37	0.5	0.38	0.38	0.39	0.38	0.27	0.47	0.37	0.38	0.55	0.59	0.47
MOTCP+$_{mean}$	0.59	0.58	0.6	0.66	0.58	0.62	0.54	0.52	0.47	0.62	0.59	0.66	0.61	0.73	0.6
MOTCP+$_{median}$	0.6	0.6	0.61	0.65	0.68	0.61	0.55	0.51	0.49	0.61	0.58	0.65	0.61	0.72	0.59
MOTCP+$_{max}$	0.71	0.77	0.81	0.81	0.79	0.77	0.63	0.71	0.7	0.8	0.84	0.86	0.69	0.87	0.71

7.4 Results

In the following subsections, we show obtained results by grouping them for each of considered criteria.

7.4.1 Effectiveness

Tables 6-10 (column $APFD_{all}$ of Traceability Links) report collected APFD values for the 21 applications object of our experiment, in presence of recovered traceability links (top) and considering all injected faults. These tables report results in terms of minimal, median, mean, and maximal APFD values achieved for each technique. In Table 11 (columns all), we report the number of times (i.e., applications) in which each technique outperformed others. To perform this comparison, we considered: *(i)* mean and median values (columns on the left), thus limiting the impact of possible outliers, *(ii)* and maximum values (columns on the right). For example, Rand achieved the highest mean and median values for APFD in one case (i.e., Woden) considering all injected faults. MOTCP+ obtained the best results for 7 applications. In some cases, more than one technique obtained the best mean and median values for APFD, so justifying why the sum of values in each column (i.e., all, ftype1, and ftype2) is greater than 21.

TABLE 8
APFD of JXPath, CommonsIO and CommonsBcel, CommonsBeanUtils and XMLGraphics

APFD															
	JXPath			CommonsIO			CommonsBcel			CommonsBeanUtils			XMLGraphics		
	all	$ftype1$	$ftype2$	all	$ftype1$	$ftype2$	all	$ftype1$	$ftype2$	all	$ftype1$	$ftype2$	all	$ftype1$	$ftype2$
Traceability Links															
Rand$_{min}$	0.45	0.26	0.41	0.36	0.32	0.2	0.37	0.29	0.29	0.44	0.24	0.27	0.38	0.28	0.24
Rand$_{mean}$	0.53	0.52	0.57	0.52	0.54	0.5	0.51	0.51	0.47	0.56	0.48	0.51	0.49	0.49	0.48
Rand$_{median}$	0.53	0.52	0.57	0.52	0.55	0.47	0.51	0.53	0.46	0.56	0.5	0.53	0.5	0.48	0.48
Rand$_{max}$	0.63	0.67	0.77	0.65	0.81	0.85	0.63	0.71	0.8	0.63	0.7	0.74	0.61	0.76	0.68
CodeCov	0.54	0.57	0.49	0.59	0.63	0.69	0.48	0.39	0.56	0.52	0.54	0.61	0.55	0.45	0.5
AddCodeCov	0.55	0.63	0.47	0.52	0.54	0.42	0.49	0.59	0.45	0.75	0.9	0.67	0.49	0.42	0.53
NSGAIIdim2 $_{min}$	0.51	0.43	0.54	0.45	0.31	0.41	0.5	0.32	0.32	0.54	0.42	0.54	0.48	0.39	0.4
NSGAIIdim2 $_{mean}$	0.6	0.59	0.63	0.52	0.47	0.55	0.58	0.5	0.65	0.61	0.51	0.65	0.55	0.55	0.57
NSGAIIdim2 $_{median}$	0.6	0.62	0.63	0.5	0.45	0.52	0.58	0.53	0.65	0.61	0.51	0.69	0.53	0.54	0.55
NSGAIIdim2 $_{max}$	0.73	0.83	0.76	0.62	0.73	0.73	0.66	0.64	0.85	0.65	0.6	0.75	0.68	0.72	0.84
MOTCP$_{min}$	0.44	0.33	0.36	0.46	0.42	0.34	0.39	0.39	0.41	0.5	0.28	0.33	0.4	0.37	0.25
MOTCP$_{mean}$	0.55	0.54	0.6	0.56	0.56	0.53	0.52	0.53	0.51	0.6	0.51	0.57	0.54	0.62	0.51
MOTCP$_{median}$	0.54	0.54	0.6	0.57	0.59	0.51	0.52	0.52	0.54	0.59	0.51	0.58	0.54	0.62	0.53
MOTCP$_{max}$	0.7	0.76	0.79	0.6	0.66	0.74	0.65	0.65	0.67	0.7	0.78	0.71	0.7	0.83	0.84
MOTCP+ $_{min}$	0.41	0.46	0.39	0.41	0.44	0.33	0.37	0.33	0.22	0.49	0.35	0.36	0.45	0.37	0.3
MOTCP+ $_{mean}$	0.56	0.62	0.6	0.54	0.62	0.54	0.53	0.48	0.5	0.61	0.52	0.57	0.55	0.56	0.58
MOTCP+ $_{median}$	0.57	0.63	0.58	0.53	0.63	0.54	0.53	0.47	0.49	0.6	0.52	0.55	0.55	0.56	0.62
MOTCP+ $_{max}$	0.67	0.81	0.88	0.68	0.79	0.73	0.68	0.73	0.76	0.71	0.68	0.82	0.71	0.78	0.89
Incomplete Traceability Links															
MOTCP+ $_{min}$	0.47	0.28	0.38	0.41	0.44	0.44	0.41	0.33	0.22	0.5	0.3	0.3	0.45	0.44	0.24
MOTCP+ $_{mean}$	0.56	0.57	0.62	0.55	0.54	0.51	0.5	0.5	0.53	0.6	0.51	0.57	0.51	0.56	0.44
MOTCP+ $_{median}$	0.56	0.58	0.63	0.53	0.63	0.5	0.53	0.47	0.49	0.59	0.5	0.58	0.55	0.56	0.44
MOTCP+ $_{max}$	0.65	0.79	0.76	0.68	0.79	0.62	0.68	0.73	0.76	0.72	0.78	0.76	0.71	0.69	0.68

TABLE 9
APFD of XMLSecurity, CommonsCollections, Pmd and CommonsLang

APFD												
	XMLSecurity			CommonsCollections			Pmd			CommonsLang		
	all	$ftype1$	$ftype2$	all	$ftype1$	$ftype2$	all	$ftype1$	$ftype2$	all	$ftype1$	$ftype2$
Traceability Links												
Rand$_{min}$	0.33	0.26	0.22	0.43	0.27	0.37	0.43	0.34	0.32	0.46	0.37	0.33
Rand$_{mean}$	0.5	0.47	0.52	0.54	0.53	0.56	0.52	0.56	0.46	0.58	0.58	0.59
Rand$_{median}$	0.5	0.46	0.47	0.55	0.52	0.55	0.52	0.56	0.43	0.58	0.59	0.58
Rand$_{max}$	0.63	0.66	0.76	0.64	0.54	0.82	0.63	0.77	0.69	0.66	0.7	0.8
CodeCov	0.52	0.61	0.52	0.66	0.54	0.49	0.54	0.3	0.69	0.54	0.66	0.54
AddCodeCov	0.51	0.67	0.48	0.52	0.56	0.8	0.63	0.6	0.61	0.54	0.31	0.59
NSGAIIdim2 $_{min}$	0.38	0.47	0.35	0.51	0.44	0.54	0.43	0.51	0.4	0.46	0.46	0.35
NSGAIIdim2 $_{mean}$	0.51	0.6	0.49	0.55	0.5	0.65	0.54	0.62	0.46	0.54	0.54	0.48
NSGAIIdim2 $_{median}$	0.53	0.64	0.49	0.55	0.49	0.65	0.54	0.61	0.46	0.54	0.47	0.46
NSGAIIdim2 $_{max}$	0.58	0.69	0.67	0.62	0.58	0.77	0.58	0.78	0.51	0.63	0.64	0.58
MOTCP$_{min}$	0.38	0.34	0.26	0.35	0.41	0.41	0.37	0.33	0.23	0.51	0.47	0.4
MOTCP$_{mean}$	0.5	0.55	0.56	0.55	0.59	0.6	0.49	0.54	0.47	0.61	0.62	0.58
MOTCP$_{median}$	0.51	0.53	0.47	0.54	0.6	0.61	0.49	0.55	0.48	0.6	0.63	0.55
MOTCP$_{max}$	0.58	0.73	0.77	0.69	0.76	0.76	0.59	0.7	0.67	0.71	0.82	0.82
MOTCP+ $_{min}$	0.4	0.22	0.2	0.42	0.28	0.2	0.41	0.33	0.36	0.51	0.42	0.45
MOTCP+ $_{mean}$	0.52	0.49	0.47	0.53	0.5	0.57	0.55	0.56	0.59	0.61	0.61	0.65
MOTCP+ $_{median}$	0.52	0.48	0.47	0.53	0.51	0.57	0.55	0.57	0.59	0.59	0.53	0.66
MOTCP+ $_{max}$	0.66	0.65	0.64	0.69	0.76	0.84	0.65	0.78	0.75	0.73	0.78	0.85
Incomplete Traceability Links												
MOTCP+ $_{min}$	0.3	0.22	0.23	0.41	0.28	0.32	0.41	0.33	0.33	0.49	0.43	0.4
MOTCP+ $_{mean}$	0.5	0.54	0.51	0.53	0.53	0.62	0.52	0.55	0.49	0.6	0.58	0.62
MOTCP+ $_{median}$	0.48	0.47	0.46	0.55	0.51	0.64	0.55	0.57	0.57	0.6	0.57	0.63
MOTCP+ $_{max}$	0.65	0.65	0.78	0.69	0.76	0.87	0.65	0.78	0.78	0.72	0.75	0.8

On the basis of values reported in Tables 6-10 and Table 11, we can observe that MOTCP+ tends to outperform other techniques. APFD values for MOTCP+ are slightly better. The results achieved by CodeCov and Rand are worse and tend to have high variability (APFD values vary in the range: $0.19 \div 0.83$), with respect to those achieved by other techniques. This trend is not statistically confirmed by the Mann-Whitney test results (see Table 12). In particular, results suggest a statistically significant difference between MOTCP+ and MOTCP (p-value < 0.001) also applying the Benjamini-Hochberg correction. It is worth mentioning that paired comparisons not listed in Table 12 have all p-values greater than 0.05. Overall, results suggest that our approach improves MOTCP by increasing the capability of test case orderings in early revealing faults and tends to outperform other approaches.

The results of a two-way permutation test suggest that observed outcomes depend on the applications object of our experiment (p-value$<$ 0.001). In particular, we noted a not trivial variability of results for all the techniques (on average 30%). For MOTCP+ and CodeCov, this variability was 34% and 38%, respectively. Results were comparable and less variable for Rand, MOTCP, NSGAIIdim2 and AddCodeCov (less than 30%). In addition, we noted that for a few applications (e.g., Log4J, iTrust, LaTazza, DBUtils) considered prioritization techniques achieved results very

TABLE 10
APFD of Jabref and Xerces

APFD	Jabref			Xerces		
	all	$ftype1$	$ftype2$	all	$ftype1$	$ftype2$
Traceability Links						
Rand_{min}	0.47	0.38	0.26	0.4	0.25	0.37
Rand_{mean}	0.57	0.61	0.58	0.55	0.5	0.48
Rand_{median}	0.56	0.59	0.48	0.56	0.47	0.53
Rand_{max}	0.67	0.74	0.82	0.63	0.84	0.71
CodeCov	0.43	0.38	0.38	0.52	0.61	0.5
AddCodeCov	0.43	0.38	0.38	0.63	0.46	0.72
NSGAIIdim2 $_{min}$	0.5	0.52	0.41	0.49	0.35	0.35
NSGAIIdim2 $_{mean}$	0.54	0.62	0.55	0.53	0.54	0.48
NSGAIIdim2 $_{median}$	0.54	0.64	0.53	0.53	0.54	0.48
NSGAIIdim2 $_{max}$	0.58	0.68	0.7	0.61	0.75	0.65
MOTCP_{min}	0.51	0.51	0.47	0.41	0.23	0.28
MOTCP_{mean}	0.59	0.64	0.63	0.54	0.45	0.51
MOTCP_{median}	0.55	0.55	0.48	0.54	0.42	0.52
MOTCP_{max}	0.6	0.65	0.54	0.67	0.75	0.69
MOTCP+ $_{min}$	0.49	0.37	0.34	0.51	0.33	0.43
MOTCP+ $_{mean}$	0.62	0.66	0.68	0.6	0.52	0.58
MOTCP+ $_{median}$	0.62	0.54	0.46	0.59	0.54	0.58
MOTCP+ $_{max}$	0.74	0.77	0.82	0.71	0.68	0.75
Incomplete Traceability Links						
MOTCP+ $_{min}$	0.51	0.37	0.52	0.49	0.38	0.4
MOTCP+ $_{mean}$	0.59	0.6	0.55	0.56	0.58	0.52
MOTCP+ $_{median}$	0.58	0.54	0.61	0.56	0.55	0.52
MOTCP+ $_{max}$	0.65	0.77	0.76	0.64	0.79	0.64

TABLE 11
Summary of results for best *APFD* results

	Mean and Median			Maximum		
	all	$ftype1$	$ftype2$	all	$ftype1$	$ftype2$
Rand	1	2	0	0	5	4
CodeCov	5	1	4	0	0	0
AddCodeCov	4	8	4	1	1	0
NSGAIIdim2	6	4	4	3	1	1
MOTCP	3	4	3	1	6	3
MOTCP+	7	5	6	17	12	15

different (20 to 25 points).

7.4.2 Sensitivity

Tables 6-10 (columns $APFD_{ftype1}$ and $APFD_{ftype2}$) report collected APFD measures for all the applications, in presence of all automatically recovered traceability links and considering severe faults (column $APFD_{ftype1}$) and faults related to related to relevant requirements (column $APFD_{ftype2}$). MOTCP+ tends to outperform other techniques for both $APFD_{ftype1}$ and $APFD_{ftype2}$ in most of the applications as descriptive statistics suggest (see descriptive statistics reported in Tables 6-10 and summary in Table 11). For example, MOTCP+ outperforms, or at least achieves comparable results, on the following applications: AveCalc, LaTazza, DBUtils, CommonProxy, iTrust, Woden, Log4J, Betwixt, JXPath, CommonsIO, XMLGraphics, and CommonsLang. On other applications, there is not a clear winner even if often either MOTCP or NSGAIIdim2 seems to be slightly better than others. Results of the Mann-Whitney test suggest that a significant difference exists between MOTCP+ and both MOTCP and Rand in terms of $APFD_{ftype1}$ as well as $APFD_{ftype2}$ (p-values are 0.01, 0.01 and 0.008, 0.04, respectively). On the other hand, no statistical significant difference was observed between MOTCP+ and both AddCodeCov and CodeCov even if a trend in favor of MOTCP+ is present. Note that pairs of other considered techniques (e.g., AddCodeCov vs. Rand) not listed in the Table 12 have p-values greater than 0.05. These results suggest that the application of MOTCP+ allows the identification of test case orderings with a higher severity and relevance with respect to baseline approaches. The results of a two-way permutation test seem to confirm the fact that $APFD_{ftype1}$ and $APFD_{ftype2}$ values depend on the application on which test case ordering techniques have been applied. That is, achieved outcomes significantly depend on considered applications (p-value < 0.001).

TABLE 12
APFD: Mann-Whitney results (in bold values significant at 5%, while * indicates values still significant by applying the Benjamini-Hochberg correction)

	$APFD_{all}$	$APFD_{ftype1}$	$APFD_{ftype2}$
MOTCP+ vs. Rand	**0.02**	**0.01***	**0.008***
MOTCP+ vs. CodeCov	0.05	0.12	0.14
MOTCP+ vs. AddCodeCov	0.59	0.74	0.16
MOTCP+ vs. NSGAIIdim2	0.093	0.096	**0.02**
MOTCP+ vs. MOTCP	<**0.001***	**0.01***	**0.041**

7.4.3 Robustness

Tables 6-10 show also collected APFD measures obtained by MOTCP+ in presence of incomplete traceability links. MOTCP+ preserves the capability in early detecting faults considering incomplete traceability links. In fact, we observed that only in a few cases MOTCP+ decreases its capability of early detecting faults. For instance, in case of $APFD_{all}$, the difference between the APFD values obtained by MOTCP+ using complete or incomplete traceability links is on average less than 10 points. That is, the overall result for Robustness suggests that the capability of defining adequate test case orderings of MOTCP+ is quite robust with respect to the goodness of traceability links.

7.4.4 Analysis of the Pareto front's metrics

Figure 4 shows examples of generated Pareto fronts for AveCalc, CommonsBcel, and CommonsCollections. Similar plots have been obtained for other applications and for all measured APFDs.

By applying the Spearman's rank correlation on the metric values obtained for each Pareto front, we observed that collinearity does not hold for the three metrics used to build the front. In fact, we observed high collinearity (>85% of correlation) only in three cases: *(i)* CommonsBeanUtils, between AUCcode and AUCcost and *(ii)* JabRef and CommonsBcel, between AUCcode and AUCreq.

Results of the PCA analysis are summarized in Table 13. This table reports: the amount of variance accounted by identified principal components (column Var); how AUCcode, AUCreq, and AUCcost contribute to these principal components (columns PC); and the corresponding *loading* value (column Load, values range in between 0 and 1 and represent the impact of a metric on a given component). For instance, 91.3% of the variance for Betwixt is explained by

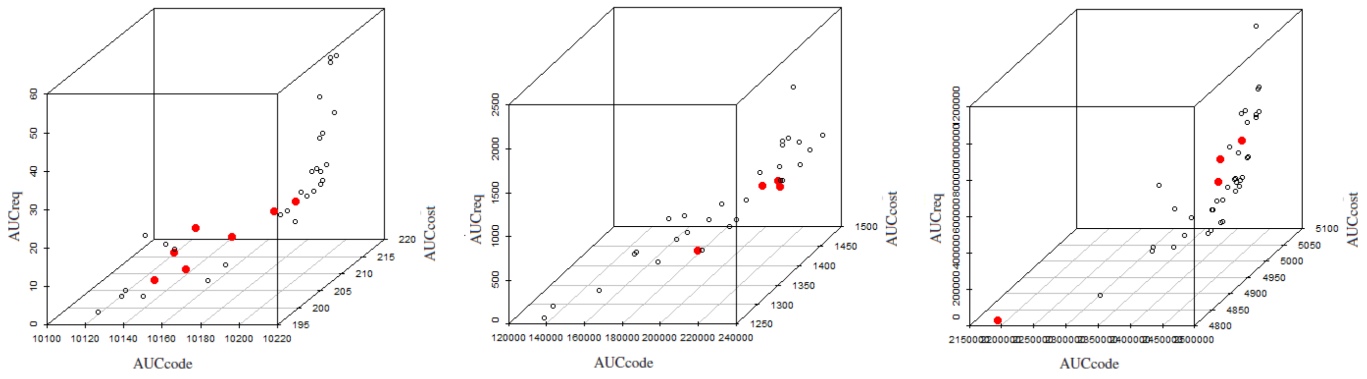


Fig. 4. Pareto fronts of $APFD_{all}$ for AveCalc, CommonsBcel and CommonsCollections. Filled circle represent test suites having higher APFD

TABLE 13
PCA summary of results for each application

Application	**Var**	**AUCcode**		**AUCreq**		**AUCcost**	
	%	PC	Load	PC	Load	PC	Load
AveCalc	77.3	1	0.6	1	0.5	1	0.5
Betwixt	91.3	1	0.6	2	0.8	1	0.6
CommonsBcel	85	1	0.6	1	0.5	1	0.5
CommonsBeanUtils	79	1	0.6	1	0.5	1	0.6
CommonsCodec	88.4	1	0.7	2	0.9	1	0.7
CommonsCollections	89.2	1	0.6	2	0.8	1	0.6
CommonsIO	91.7	-	-	2	0.72	1	0.67
CommonsLang	94.7	1	0.71	-	-	2	0.73
DBUtis	89.2	2	0.95	1	0.68	1	0.71
iTrust	94.6	1	0.69	2	0.87	1	0.71
Jabref	78.1	1	0.62	1	0.6	1	0.49
JTidy	92.1	1	0.67	2	0.9	1	0.62
JXPath	95.9	1	0.66	2	0.94	1	0.7
LaTazza	97.6	-	-	1	0.97	2	0.97
Log4J	94.6	1	0.64	2	0.89	1	0.71
Pmd	91.8	1	0.72	2	0.92	1	0.67
Woden	93.2	1	0.71	2	0.78	-	-
Xerces	74.7	1	0.61	1	0.6	1	0.51
XMLGraphics	92.7	1	0.73	2	0.75	-	-
XMLSecurity	92.9	1	0.71	2	0.96	1	0.69
CommonsProxy	93.7	1	0.7	2	0.99	1	0.7

the first two principal components (i.e., in the columns PC for Betwixt we can see 1 and 2 representing two principal components); AUCcode and AUCcost mainly load on the first component (the value of columns PC these two metrics is 1), while AUCreq mainly loads on the second component (the value of column PC for this metric is 2). All the three metrics do not have a trivial impact on components (their Load value is $\geq$0.5 for all metrics). A similar trend is shown for most of the other applications, but for a few of them (e.g., CommonsIO, LaTazza) not all metrics significantly load on principal components (see the symbol - in column PC). In the case of CommonsIO, for example, AUCcost and AUCreq load respectively on the first and the second component, instead AUCcode does not load on a specific component. Overall, results in table confirms that our three metrics contribute to principal components with a clear impact and that a trend exists for which such metrics share a conceptual meaning on their impact on the components: AUCcode and AUCcost seem to refer to software execution, while AUCreq to software specification.

In Figure 4, filled circles represent the best solutions (those having at least 80% of the maximum APFD in the front) we found in three Pareto fronts, that is test suites having higher APFD values in the Pareto Fronts. In Table 14, we summarize the distribution of these best solutions in their respective fronts, where each axis of the front has been divided by three with the aim of identifying three areas in the front having respectively: low, medium, and high AUC value. The results suggest that most of best solutions have high values of AUCcode (71% of the best solutions), AUCreq (85% of them), and AUCcost (66% of them). In other words, best solutions can be frequently found in the top-right part of obtained Pareto Front.

7.4.5 *Impact of application objects and artifacts*

We analyzed the impact of some aspects of both applications and used artifacts. In particular, we considered: *(i)* size (i.e., size of the considered applications, number of requirements, and size of test suites); *(ii)* distribution of injected faults (e.g., number of faults injected in code that implements a requirement, number of requirements not tested by any test case, and density of the faults per requirements); and *(iii)* capability of test cases in revealing faults (e.g., number of test cases revealing one fault, number of test cases revealed one or more than one fault, number of test cases that reveal two or more than three faults, and functional test case redundancy).

In Table 15, we summarize results concerning how injected faults impact on application requirements. The table reports (second column) the percentage of requirements affected by at least one fault. For instance, 40% (i.e., 4 out 10) of requirements considered for AveCalc were affected by at least one fault. Moreover, this table also reports (third column) the percentage of test cases that do not impact considered requirements. For examples, 3 out of 20 faults (15%) of CommonsCollections did not impact on the set of considered application requirements. The percentage of not tested requirements is reported in the fourth column. In the fifth column, the table shows fault density ($FaultDensity = \sum_{r \in req}(\frac{|NumFaults_r - \text{mean}(\frac{NumFaults}{NumReqs})|}{NumReqs})$). High values of fault density indicate an application in which faults are concentrated in a few requirements of the application, while a low level represents an application

TABLE 14
Distribution of best solutions (higher APFD) in the Pareto Fronts

Application	**Best**	**AUCcode**			**AUCreq**			**AUCcost**		
	Solutions	High	Medium	Low	High	Medium	Low	High	Medium	Low
AveCalc	7	2	1	**4**	**4**	3	0	**7**	0	0
Betwixt	3	**2**	1	0	**3**	0	0	1	0	**2**
CommonsBcel	4	**3**	1	0	**4**	0	0	**4**	0	0
CommonsBeanUtils	6	**6**	0	0	**4**	2	0	0	**3**	**3**
CommonsCodec	3	**3**	0	0	1	**2**	0	**2**	1	0
CommonsCollections	4	**3**	0	1	**2**	0	**2**	**2**	**2**	0
CommonsIO	3	**2**	1	0	**3**	0	0	0	**2**	1
CommonsLang	3	0	**2**	1	**2**	1	0	**2**	1	0
CommonsProxy	2	**2**	0	0	**1**	0	**1**	0	**1**	**1**
DBUtils	4	**2**	0	**2**	**3**	0	1	**2**	1	1
iTrust	1	**1**	0	0	**1**	0	0	0	**1**	0
Jabref	3	**3**	0	0	**3**	0	0	**2**	1	0
JTidy	6	**4**	0	2	2	**3**	1	**5**	0	1
JXPath	9	4	**5**	0	**7**	2	0	**4**	**4**	1
LaTazza	2	0	**2**	0	**1**	0	**1**	**1**	**1**	0
Log4J	10	**7**	3	0	**7**	2	1	2	**6**	2
Pmd	7	**5**	0	2	**4**	2	1	2	**4**	1
Woden	3	0	1	**2**	**3**	0	0	**3**	0	0
Xerces	3	**3**	0	0	**3**	0	0	**3**	0	0
XmlGraphics	6	**2**	**2**	**2**	**3**	**3**	0	**5**	1	0
XmlSecurity	4	1	1	**2**	1	1	**2**	**2**	1	1
	-	15(71%)	4 (19%)	5(23%)	18(85%)	3(14%)	4(19%)	14(66%)	9(42%)	3(14%)

TABLE 15
Impact of faults on requirements.

Application	**Reqs affected by fault (%)**	**Faults not impacting reqs (%)**	**Not tested reqs (%)**	**Fault Density**	**Req 80% faults (%)**
LaTazza	60	13	40	1.8	40
AveCalc	40	0	10	3.2	30
CommonsProxy	80	0	10	1.7	40
DBUtis	58	14	25	1.6	50
iTrust	80	23	0	1.8	40
CommonsCodec	57	35	10	2	31
JTidy	36	6	8	1.7	24
Woden	41	21	0	1.8	29
Log4J	41	0	4	1.7	16
Betwixt	77	10	0	1.8	33
JXPath	55	15	0	2	30
CommonsIO	50	0	0	3.4	22
CommonsBcel	25	0	0	1.6	10
CommonsBeanUtils	57	22	7	1.8	30
XMLGraphics	50	20	12	4.3	12
XMLSecurity	13	0	0	1.9	39
CommonsCollections	41	15	5	1.8	29
Pmd	50	10	10	2	25
CommonsLang	80	5	0	2.2	31
Jabref	51	50	22	1.5	32
Xerces	80	0	0	1.6	30

TABLE 16
Percentage of test cases revealing: at least one fault, only one fault, and more than one fault for each application; and functional test case redundancy

Application	**TCS revealing $\geq$1 fault**	**TCS revealing =1 fault**	**TCS revealing >1 fault**	**TCS Redundancy**
LaTazza	68	17	51	8.2
AveCalc	53	4	49	11.7
CommonsProxy	8	7.8	0.2	5.6
DBUtils	8	7	1	10.5
iTrust	4	3.6	0.4	3.6
CommonsCodec	5	4.9	0.1	13.8
JTidy	4	3.7	0.3	14.4
Woden	8	7.2	0.8	4.2
Log4j	4	3.1	0.9	21
Betwix	9	8.4	0.6	2.9
JXPath	7	6.6	0.4	6.5
CommonsIO	3	2.8	0.2	10.8
CommonsBcel	26	24	2	4.1
CommonsBeanUtils	3	2.8	0.2	15.8
XMLGraphics	8	8	0	3.3
XMLSecurity	18	18	0	3.2
CommonsCollections	3	2.8	0.2	4.3
Pmd	3	2.9	0.1	3.8
CommonsLang	1.4	1.1	0.3	20
Jabref	13	13	0	9.4
Xerces	7	6.3	0.7	8.5

in which faults are spread among many requirements. In the last column, we report the percentage of requirements in which 80% of faults have been injected, e.g., 80% of faults in LaTazza have been injected into 40% (i.e., 4) requirements out of 10 considered.

From Table 15, we can also observe that injected faults were evenly distributed among application requirements (i.e., distributed in more than 51% of requirements — median value for Reqs affected by faults — and with a fault density lower or equal than 1.8 – median value for FaultDensity) in case of: LaTazza, CommonsProxy, DBUtils, iTrust, Betwixt, CommonsBeanUtils, and Xerces. Conversely, injected faults seem to be concentrated in a few requirements (i.e., distributed in less than 51% of requirements and with a fault density higher or equal than 1.8) in: AveCalc, Woden, CommonsIO, XMLSecurity, and CommonsCollections.

On the base of results shown in Table 15 (fourth column), for LaTazza and DBUtils a high number of requirements, 40% (4 out of 10) and 25% (3 out of 12) respectively, were not linked to any test case. This indicates that test cases are mainly focused on a subset of considered requirements. As for iTrust, Woden, Betwixt, JXPath, CommonsIO, CommonsBcel, XMLSecurity, CommonsLang, and Xerces, all the requirements were linked to at least one test case, while for the other applications few requirements (on average 7.7% for each application) were not linked with test cases, even if some links were present. These results suggest that the set of used traceability links could be incomplete.

Results reported in Table 16 (second column) suggest that the test suites of AveCalc, LaTazza, CommonsBcel, and XMLSecurity have 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) have less than 5% of fault-revealing test cases. For each application, the third and the fourth column of Table 16 show the percentage of test cases revealing only one and more than one fault, respectively. We can see that: a limited percentage of test cases (less than 25%) of considered suites reveals one fault. Only in case of AveCalc and LaTazza, a large percentage of test cases (about 50%) reveal more than one fault, while in the remaining applications almost all test cases reveal only one of injected faults.

In the last column of Table 16, results for test case redundancy (TCSRedundancy) are shown. This measure is computed as follows: $\frac{|TCS|}{|TestClasses|}$. TCS is the number of test cases composing a test suite and $TestClasses$ represents the JUnit classes that functionally group test cases. We assume that JUnit classes group functionally correlated test cases, i.e., JUnit test methods. Results suggest that the test suites with high redundancy are those of: Log4j, CommonsLang, CommonsBeanUtils, JTidy, and CommonsCodec.

In Table 17, we summarize the results of a two-way permutation test on considered co-factors and their interaction. Results suggest that there is a significant effect of application with respect to APFD$_{all}$ values. Moreover, other factors that have shown some influence on experimental results are: capability of revealing faults of used test cases in terms of test cases that reveal only one fault (PercTcsRevealingOneBug); number of test cases composing test suites (NumberOfTCS) as well as number of requirements (NumReqs); number of requirements containing 80% of injected faults (PercReq80%faults) as well as fault density (FaultDensity); and test case redundancy (TCSRedundancy). By correlating such metrics with APFD values (using the Spearman's Rank Correlation Coefficient) we found relevant and statistical impacts for: PercTcsRevealingOneBug toward Rand (ρ=0.14) and MOTCP (ρ=-0.08), while in case of NumReqs, PercReq80%faults, FaultDensity and TCSRedundancy toward Rand (ρ=-0.23, ρ=0.26, ρ=-0.09 and ρ=0.23), MOTCP (ρ=-0.13, ρ=0.14, ρ=-0.07 and ρ=0.4) and MOTCP+ (no correlation, ρ=0.09, ρ=-0.13 and ρ=0.3). Notice that CodeCov and AddCodeCov do not have any correlation with these co-factors.

7.4.6 Additional analysis

We performed an additional analysis to study possible overhead of our proposal with respect to baseline approaches. In Table 18, we report some descriptive statistics (i.e., minimal, median, maximal, mean, and standard deviation values) of the overall time for prioritizing test cases by applying our proposal and baseline approaches on the studied applications. In the experimentation, we used a PC equipped by 2.20 GHz Intel Core i7 with 8 GB of RAM and Windows 8 (64-bit) as operating system.

MOTCP, NSGAIIdim2, and MOTCP+ required a comparable time to prioritize test cases and CodeCov and Rand were faster with respect to other approaches. As for AddCodeCov, we observed that it is either fast or slow to prioritize test cases. This seems to depend on the application. In particular, we noted that for medium to large applications (e.g., CommonsLang and CommonsBeanUtils), AddCodeCov required more time than other approaches. In the case of CommonsLang, MOTCP+ required 580.9 seconds, while AddCodeCov required 1794.2 seconds.

Another result of our analysis is that MOTCP and MOTCP+ required more time than other approaches because of the time needed to recover traceability links. This time is, on average, 36% of the overall time required to multi-objective algorithms to get final test case prioritization. However, we can postulate that the time to recover traceability links is hidden to the user if the recovery process is executed in background every time requirements, test cases, or source code are modified. For such a reason,

TABLE 17
Two-way permutation test on the relevant co-factors

Factor	**p-value**
Technique	< **0.001**
Application	< **0.001**
Technique:Application	< **0.001**
Technique	<**0.001**
PercTcsRevealingAtLeastOneBug	< **0.001**
Technique:PercTcsRevealingAtLeastOneBug	1
Technique	<**0.001**
PercTcsRevealingOneBug	**0.127**
Technique:PercTcsRevealingOneBug	**0.001**
Technique	< **0.001**
PercTcsRevealingMoreThanOneBug	< **0.001**
Technique:PercTcsRevealingMoreThanOneBug	0.452
Technique	< **0.001**
PercReqAffectedByAtLeast1Bug	< **0.001**
Technique:PercReqAffectedByAtLeast1Bug	0.122
Technique	< **0.001**
PercFaultNotImpactingReq	< 0.**001**
Technique:PercFaultNotImpactingReq	0.851
Technique	< **0.001**
AppSize	**1**
Technique:AppSize	0.5781
Technique	**0.039**
NumberOfTCS	**0.047**
Technique:NumberOfTCS	**0.005**
Technique	< **0.001**
NumReqs	< **0.001**
Technique:NumReqs	< **0.001**
Technique	**0.048**
PercReqNonLinkedToTCS	< **0.001**
Technique:PercReqNonLinkedToTCS	1
Technique	< **0.001**
PercReq80%faults	**0.065**
Technique:PercReq80%faults	**0.006**
Technique	< **0.001**
FaultDensity	< **0.001**
Technique:FaultDensity	< **0.001**
Technique	< **0.001**
TCSRedundancy	< **0.001**
Technique:TCSRedundancy	< **0.001**

TABLE 18
Descriptive statistics of the overall execution time (in seconds) of prioritization approaches

	MOTCP+	**MOTCP**	**NSGAIIdim2**	**AddCodeCov**	**CodeCov**	**Rand**
Min	14.8	13.9	14.0	0.8	0.8	0.0
Mean	160.5	161.3	162.0	48.6	20.9	1.2
Median	319.7	317.6	318.6	150.4	41.7	2.0
Max	2160.0	2026.4	2022.2	1794.2	297.5	9.9
StDev	488.7	464.6	463.8	387.1	65.3	2.5

TABLE 19
Time to recover links between requirements and code and requirements and test cases

	Reqs - TestCases	Reqs - Code
Min	2.0	10.0
Mean	163.4	49.9
Median	90.0	25.0
Max	1505.0	430.0
StDev	320.5	91.5

we report in Table 19 some descriptive statistics on the time to recover traceability links. In particular, the second column reports the time to recover traceability links between requirements and test cases, while the third column reports the time to recover links between requirements and source code. As shown in Table 19, the recovery of traceability links requirements and test cases is more expensive since it required on average 163.4 seconds. The recovery of traceability links between requirements and source code required on average 49.9 seconds. We argue that this is due to the kind of artifacts on which LSI was applied. It is useful to observe that in a real project, requirements and test cases change less frequently than source code.

7.4.7 *Analysis*

We summarize achieved results, their interpretation, and observed trends as follows:

- **Capability to find faults**. By considering the three sets of faults used to evaluate both effectiveness and sensitivity, MOTCP+ mostly outperforms other techniques. In detail, by considering the median of APFD values we see that MOTCP+ outperforms: *(i)* MOTCP 80% of the applications, *(ii)* AddCodeCov 66.6% of the applications, and *(iii)* NSGAIIdim2 62% of the applications. Only for 2 applications (i.e., CommonsProxy and iTrust) the median value achieved by MOTCP+ is lower than the one of baselines and, in particular, of AddCodeCov, NSGAIIdim2 and MOTCP. Hence, by trying to balance between low- and high-level information, MOTCP+ tends to outperform the traditional multi-objective technique based on two dimensions (code-coverage and execution time). By applying automatic weighting, it seems that MOTCP+ is more efficient than MOTCP in finding faults. In fact, test orderings produced by MOTCP+ in almost all cases are better that those produced by MOTCP. This could be mainly due to distribution of the faults. In this concern, we observed that AddCodeCov and MOTCP tend to achieve better results if faults are evenly spread in a high number of application requirements. MOTCP+ tends to achieve better results if faults are concentrated in a few requirements. NSGAIIdim2 seems quite stable with respect to fault distribution. These findings seem to be in line with our initial hypothesis about the use of automatic weighting of application code and requirements to give more relevance to specific and fault-prone portions of the application. Indeed, we measured a positive correlation (Spearman's Rank correlation coefficient is equal to 0.36 and p-value$<$0.001) between requirements rankings obtained by applying automatic weighting and distribution of injected faults into requirements. In other terms, our metric-based automatic weighting approach is reasonably able to identify fault-prone requirements.
- **Robustness**. MOTCP+ seems to be able to support a limited amount of spurious traceability links. That is, the quality of traceability links might affect ordering results even if not in significant way.
- **Pareto's metrics**. All three considered metrics seem to have a relevant impact on our results. In a few applications, we observed that two metrics could be considered instead of three without loosing information. In general, we observed that all the metrics significantly contributed to test case ordering results in terms of high values of AUCcode, AUCreq, and AUCcost.
- **Co-factors**. As for the APFD values, we observed that results can vary with respect to the studied applications, namely our experimental objects. In particular, a variance around 30% for all techniques was observed. Test suite composition (e.g., percentage of test cases revealing one or more than one fault, percentage of requirements linked to test cases, and test case redundancy) and fault distribution can impact on achieved results. In particular, we see that fault density has only a (negative) limited impact on results achieved by our approach. This result suggests that our approach can achieve reasonably good results if faults are spread in code and requirements as well as if they are concentrated in a few requirements. In fact, AddCodeCov and MOTCP achieve better results than MOTCP+ if faults are evenly spread across a high number of requirements. However, an increase of fault density in a few requirements lets decrease capability of AddCodeCov and MOTCP in early revealing faults. NSGAIIdim2 seems to be less sensitive than other techniques to changes in bug density. Moreover, test case redundancy can increase the performance of our approach, while it does not significantly impact on AddCodeCov. Another aspect that seems to negatively impacts capability of AddCodeCov in early detecting fault is the number of test cases of a suite that discovers at least one faults: at increase of such a number test orderings generated by AddCodeCov decrease their APFD values. Instead, APFD values of multi-objective approaches tends to increase if we observed an increase of the number of test cases revealing one fault. An aspect that seems to penalize the multi-objective approaches, while it favors performance of AddCodeCov, is the number of test cases to be ordered. We indeed observed that a strong increase of the number of test cases in a test suite can decrease APFD values of test orderings obtained by the considered multi-objective approaches while the APFDs of test ordering produced by AddCodeCov increase, as well as computation time required to AddCodeCov to find final test case orderings.

7.4.8 *Implications*

We distilled the findings of our experiment adopting a perspective-based approach [62]. We focus on the practitioner/consultant (simply practitioner in the following) and researcher's perspectives [63]:

1) The results support our initial hypothesis about efficiency and effectiveness of our technique as well as about the use of automatic artifacts analysis and weighting during the prioritization of test cases. That is, test case orderings obtained by applying our approach are able to early recover faults that are both technical and business relevant. This result is relevant for the practitioner interested in using our approach in his/her company.
2) The experiment is focused on different kinds of applications and the magnitude of benefits deriving from the use of three dimensions suggests that obtained result could be also generalized in different contexts. This point deserves further investigations and it is relevant for both the practitioner and the researcher.
3) The experimental objects were realistic enough for small- to medium-sized software projects. Although we are not sure that achieved results scale to real commercial/industrial projects, the results seem to reassure us that the outcomes might be generalized to larger projects. This point is clearly relevant for the practitioner and deserves future investigations.
4) By explicitly considering functional-dimension during the test case prioritization, our technique can give more relevance to those test cases capable to reveal severe and requirement-relevant fault, thus outperforming traditional techniques that conversely tend to give the same relevance to each fault. This point is relevant for the researcher.
5) To let our test prioritization technique consider functional aspects, application artifacts (e.g., requirements and source code) have to be analyzed before doing the test case orderings definition. Hence, the collected information can let us produce more efficient test case orderings but they introduce additional and not trivial cost required to identify adequate test ordering. This aspect is clearly relevant for the practitioner interested in reducing the cost for performing regression testing and for identifying effective test case orderings. The researcher could be interested in investigating possible strategies to identify a trade-off between these two concerns.
6) From the execution time point of view, the recovery of traceability links is the most expensive part of the process underlying our approach. This aspect is particularly relevant for the researcher. In particular, the researcher could be interested in studying either different text retrieval model and technique or improving the performances of the used IR technique. The practitioner interested in our approach has to take into account the additional execution cost introduced by LSI use or has to explicitly document traceability links.
7) The diffusion of a new technology/method is made easier when empirical evaluations are performed and their results show that such a technology/method solves actual issues [64]. Therefore, results from our experiment could speedup the transferring of our solution to the industry. In addition, its introduction should not require a complete and radical process change in a given company because of the use of automatically recovered traceability links. This point has particular interest for the practitioner.

8 Conclusions

We propose a multi-objective technique to identify test case orderings that are effective (in terms of capability in early discovering faults) and efficient (in terms of execution cost). To this end, our proposal takes into account the coverage of source code and application requirements and the cost to execute test cases. An IR-based traceability recovery approach has been applied to link software artifacts (i.e., requirements specifications) with source code and test cases. A test case ordering is then determined by using a multi-objective optimization, implemented in terms of NSGA-II. The proposed technique applies a metric-based approach to automatically identify critical and fault-prone portions of software artifacts, thus becoming able to give them more importance during test case prioritization. Our technique has been validated on 21 Java applications. The most important take-away result of our experimental evaluation is: our approach is able to identify test case orderings that early recover faults both technical and business relevant.

References

[1] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2010.
[2] J. Karlsson and K. Ryan, "A cost-value approach for prioritizing requirements," *IEEE Software*, vol. 14, pp. 67–74, 1997.
[3] S. Mohanty, A. Acharya, and D. Mohapatra, "A survey on model based test case prioritization," *International Journal of Computer Science and Information Technologies*, vol. 2, no. 3, pp. 1002 – 1040, 2011.
[4] M. Islam, A. Marchetto, A. Susi, and G. Scanniello, "A Multi-Objective Technique to Prioritize Test Cases Based on Latent Semantic Indexing," in *Proc. of European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, 2012, pp. 21–30.
[5] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman, "Indexing by Latent Semantic Analysis," *Journal of the American Society of Information Science*, vol. 41, no. 6, pp. 391–407, 1990.
[6] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *Proc. of International Conference on Software Engineering*. IEEE Computer Society, 2003, pp. 125–137.
[7] A. De Lucia, M. Di Penta, and R. Oliveto, "Improving source code lexicon via traceability and information retrieval," *IEEE Transactions on Software Engineering*, vol. 37, no. 2, pp. 205 –227, march-april 2011.
[8] Z. Li, M. Harman, and R. Hierons, "Search algorithms for regression test case prioritization," *IEEE Transactions on Software Engineering*, vol. 33, no. 4, pp. 225 –237, april 2007.
[9] M. Salehie, S. Li, L. Tahvildari, R. Dara, S. Li, and M. Moore, "Prioritizing requirements-based regression test cases: A goal-driven practice," in *Proc. of European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, 2011, pp. 329 –332.
[10] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "Timeaware test suite prioritization," in *Proc. of International Symposium on Software Testing and Analysis*. ACM, 2006, pp. 1–12.
[11] S. Yoo and M. Harman, "Using hybrid algorithm for pareto efficient multi-objective test suite minimisation," *Journal of Systems and Software*, pp. 689–701, 2009.

[12] V. R. Basili and R. W. Selby, "Comparing the effectiveness of software testing strategies," *IEEE Transactions on Software Engineering*, vol. 13, pp. 1278–1296, December 1987.

[13] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE Transactions on Software Engineering*, vol. 28, pp. 159–182, February 2002.

[14] G. Rothermel, R. Untch, C. Chu, and M. Harrold, "Test case prioritization: an empirical study," in *Proc. of International Conference on Software Maintenance*. IEEE Computer Society, 1999, pp. 179 –188.

[15] D. Marijan, A. Gotlieb, and S. Sen, "Test case prioritization for continuous regression testing: An industrial case study," in *Proc. of International Conference on Software Maintenance*. IEEE Computer Society, 2013, pp. 540–543.

[16] D. Di Nardo, N. Alshahwan, L. Briand, and Y. Labiche, "Coverage-based test case prioritisation: An industrial case study," in *Proc. of International Conference on Software Testing, Verification and Validation*. IEEE Computer Society, 2013, pp. 302–311.

[17] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei, "Bridging the gap between the total and additional test-case prioritization strategies," in *Proc. of International Conference on Software Engineering*. IEEE Computer Society, 2013, pp. 192–201.

[18] S. Yoo and M. Harman, "Pareto efficient multi-objective test case selection," in *Proc. of International Symposium on Software testing and Analysis*. ACM, 2007, pp. 140–150.

[19] W. Sun, Z. Gao, W. Yang, C. Fang, and Z. Chen, "Multi-objective test case prioritization for gui applications," in *Proc. of ACM Symposium on Applied Computing*. ACM, 2013, pp. 1074–1079.

[20] A. M. Memon, M. L. Soffa, and M. E. Pollack, "Coverage criteria for gui testing," *SIGSOFT Softw. Eng. Notes*, vol. 26, no. 5, pp. 256–267, Sep. 2001.

[21] R. Kavitha, V. Kavitha, and N. Kumar, "Requirement based test case prioritization," in *Proc. of International Conference on Communication Control and Computing Technologies*, 2010, pp. 826 –829.

[22] M. Arafeen and H. Do, "Test case prioritization using requirements-based clustering," in *Proc. of International Conference on Software Testing, Verification and Validation*, March 2013, pp. 312–321.

[23] C. Nguyen, A. Marchetto, and P. Tonella, "Test case prioritization for audit testing of evolving web services using information retrieval techniques," in *Proc. of International Conference on Web Services*. IEEE Computer Society, 2011, pp. 636 –643.

[24] C. Fang, Z. Chen, K. Wu, and Z. Zhao, "Similarity-based test case prioritization using ordered sequences of program entities," *Software Quality Journal*, vol. 22, no. 2, pp. 335–361, 2014.

[25] OMG, "Unified modeling language (OMG UML) specification, version 2.3," Object Management Group, Tech. Rep., May 2010.

[26] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.

[27] A. Marchetto, C. Di Francescomarino, and P. Tonella, "Optimizing the trade-off between complexity and conformance in process reduction," in *Proc. of International Conference on Search Based Software Engineering*. Springer-Verlag, 2011, pp. 158–172.

[28] Y. Zhang and M. Harman, "Search Based Optimization of Requirements Interaction Management," in *Proc. of International Symposium on Search Based Software Engineering*. IEEE Computer Society, 2010, pp. 47–56.

[29] O. C. Z. Gotel and C. W. Finkelstein, "An Analysis of the Requirements Traceability Problem," in *Proc. of International Conference on Requirements Engineering*, 1994, pp. 94–101.

[30] S. Klock, M. Gethers, B. Dit, and D. Poshyvanyk, "Traceclipse: an eclipse plug-in for traceability link recovery and management," in *Proc. of International Workshop on Traceability in Emerging Forms of Software Engineering*. ACM, 2011, pp. 24–30.

[31] M. Lormans and A. van Deursen, "Reconstructing requirements coverage views from design and test using traceability recovery via LSI," in *Proc. of International Workshop on Traceability in Emerging Forms of Software Engineering*. ACM, 2005, pp. 37–42.

[32] A. Qusef, R. Oliveto, and A. De Lucia, "Recovering traceability links between unit tests and classes under test: An improved method," in *Proc. of International Conference on Software Maintenance*. IEEE Computer Society, 2010, pp. 1 –10.

[33] X. Zou, R. Settimi, and J. Cleland-Huang, "Improving automated requirements trace retrieval: a study of term-based enhancement methods," *Empirical Software Engineering*, vol. 15, pp. 119–146, April 2010.

[34] S. K. Sundaram, J. H. Hayes, A. Dekhtyar, and E. A. Holbrook, "Assessing traceability of software engineering artifacts," *Requirements Engineering*, vol. 15, no. 3, pp. 313–335, 2010.

[35] R. Branda, A. Tolve, L. Mazzeo, and G. Scanniello, "Linking e-mails and source code using BM25F," in *Proc. of International Conference on Software Engineering Advances*. IARIA, 2013, pp. 271–277.

[36] A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora, "Recovering traceability links in software artifact management systems using information retrieval methods," *ACM Transactions on Software Engineering and Methodology*, vol. 16, no. 4, 2007.

[37] G. Salton and M. J. McGill, *Introduction to Modern Information Retrieval*. New York: McGraw Hill, 1983.

[38] A. Abadi, M. Nisenson, and Y. Simionovici, "A traceability technique for specifications," in *Proc. of International Conference on Program Comprehension*. Washington, DC, USA: IEEE CS Press, 2008, pp. 103–112.

[39] S. Wang, D. Lo, Z. Xing, and L. Jiang, "Concern localization using information retrieval: An empirical study on linux kernel," in *Proceedings of Working Conference on Reverse Engineering*. IEEE CS Press, 2011, pp. 92–96.

[40] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Bug localization using latent dirichlet allocation," *Information & Software Technology*, vol. 52, no. 9, pp. 972–990, Sep. 2010.

[41] C. D. Manning, P. Raghavan, and H. Schütze, *An Introduction to Information Retrieval*. Cambridge University Press, England, 2009.

[42] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, Jun. 1994.

[43] M. Eaddy, T. Zimmermann, K. D. Sherwood, V. Garg, G. C. Murphy, N. Nagappan, and A. V. Aho, "Do Crosscutting Concerns Cause Defects?" *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 497–515, Jul. 2008.

[44] B. C. da Silva, C. Sant'Anna, and C. Chavez, "Concern-based cohesion as change proneness indicator: an initial empirical study," in *Proc of Workshop on Emerging Trends in Software Metrics*. ACM, 2011, pp. 52–58.

[45] E. Figueiredo, C. Sant'Anna, A. Garcia, T. T. Bartolomei, W. Cazzola, and A. Marchetto, "On the maintainability of aspect-oriented software: A concern-oriented measurement framework," in *Proc. of European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, 2008, pp. 183–192.

[46] R. E. Lopez-Herrejon and S. Apel, "Measuring and characterizing crosscutting in aspect-based programs: Basic metrics and case studies," in *Proc. of International Conference on Fundamental Approaches to Software Engineering*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 423–437.

[47] R. Lincke, J. Lundberg, and W. Löwe, "Comparing Software Metrics Tools," in *Proc. of Symposium on Software Testing and Analysis*. ACM, 2008, pp. 131–142.

[48] M. Asghar, A. Marchetto, A. Susi, and G. Scanniello, "Maintainability-Based Requirements Prioritization by Using Artifacts Traceability and Code Metrics," in *Proc. of European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, 2013, pp. 417–420.

[49] K. Atkinson, *An Introduction to Numerical Analysis*, 2nd ed. Wiley, 1989.

[50] S. Sivanandam and S.N.Deepal, *Introduction to genetic algorithms*. Springer, 2008.

[51] M. M. Islam, A. Marchetto, A. Susi, and G. Scanniello, "MOTCP: A Tool for the Prioritization of Test Cases Based on a Sorting Genetic Algorithm and Latent Semantic Indexing," in *Proc. of International Conference on Software Maintenance*. IEEE Computer Society, 2012, pp. 654–657.

[52] J. J. Durillo and A. J. Nebro, "jMetal: A Java Framework for Multi-Objective Optimization," *Advances in Engineering Software*, vol. 42, no. 10, pp. 760 – 771, 2011.

[53] V. Basili, G. Caldiera, and D. H. Rombach, *The Goal Question Metric Paradigm, Encyclopedia of Software Engineering*. John Wiley and Sons, 1994.

[54] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Springer, 2012.

[55] S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization," in *Proc. of International Conference on Software Engineering*. IEEE Computer Society, 2001, pp. 329 – 338.

[56] Y. Benjamini and Y. Hochberg, "Controlling the false discovery rate: A practical and powerful approach to multiple testing," *Journal of the Royal Statistical Society Series B (Methodological)*, vol. 57, no. 1, pp. 289–300, 1995.
[57] R. Baker, "Modern permutation test software," *In E. Edgington Randomization Tests, New York, Marcel Decker*, 1995.
[58] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proc. of International Conference on Software Engineering*. ACM, 2005, pp. 402–411.
[59] A. Marchetto and P. Tonella, "Using search-based algorithms for ajax event sequence generation during testing," *Empirical Software Engineering*, vol. 16, no. 1, pp. 103–140, 2011.
[60] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel, "The effects of time constraints on test case prioritization: A series of controlled experiments," *IEEE Transactions on Software Engineering*, vol. 36, no. 5, pp. 593–617, 2010.
[61] H. Leung and L. White, "Insights into regression testing," in *Proc. of International Conference on Software Maintenance*. IEEE Computer Society, 1989, pp. 60 – 69.
[62] V. R. Basili, S. Green, O. Laitenberger, F. Lanubile, F. Shull, L. S. Sørumgård, and M. V. Zelkowitz, "The empirical investigation of perspective-based reading," *Empirical Software Engineering*, vol. 1, no. 2, pp. 133–164, 1996.
[63] B. Kitchenham, H. Al-Khilidar, M. Babar, M. Berry, K. Cox, J. Keung, F. Kurniawati, M. Staples, H. Zhang, and L. Zhu, "Evaluating guidelines for reporting empirical software engineering studies," *Empirical Software Engineering*, vol. 13, pp. 97–121, 2008.
[64] S. L. Pfleeger and W. Menezes, "Marketing technology to software practitioners," *IEEE Software*, vol. 17, no. 1, pp. 27–33, 2000.

Alessandro Marchetto is currently an independent researcher working in the field of Software Engineering. He received his PhD degree in Software Engineering from the University of Milano, Italy in 2007. From 2006 till the end of 2012 he was a researcher at the Center for Information Technology (CIT) of the Bruno Kessler Foundation in Trento, Italy, working with the Software Engineering group. His primary research interests concern Software Engineering and, in particular, include quality, verification and testing of Software Systems and of Internet-based systems. He published more than 80 papers in primary international conferences and journals. He regularly reviews papers for international conferences (e.g., ICSE, ICSM, CSMR) and journals (e.g., TSE, IST, JSS, IET). He collaborated to the organization of more than ten international scientific events (e.g., SSBSE 2012, SCAM 2012, EmpiRE 2011-2012-2013-2014, WSE 2008-2012).

Md. Mahfuzul Islam works as a Software Developer in Create-Net and Exrade Srl. His research interests include Requirements Engineering and Software design, Software testing, and Data analytics. He completed his BSc in Computer Science and Engineering from American International University Bangladesh (Bangladesh) and MSc from University of Trento (Italy) specialized in Software Technologies. He has more than 5 years of working experience in research projects (such as Superhub and Seeinnova EU projects).

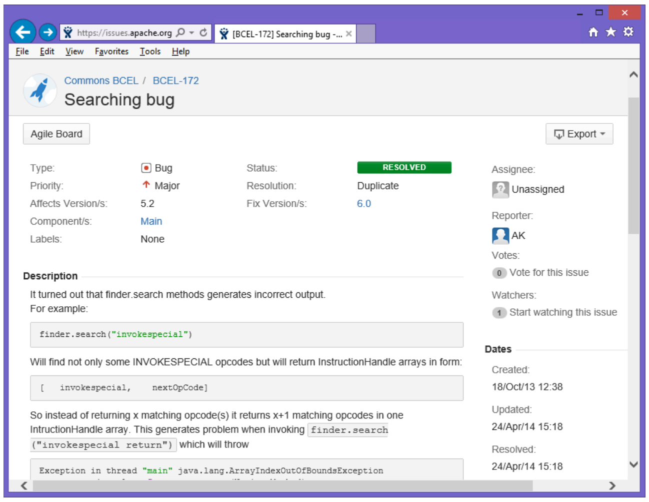
Waseem Asghar works as a Software Analyst in TVEyes Language Technology. He completed his MSc in Computer Science and Engineering in the University of Trento, Italy, discussing a thesis about prioritisation techniques in Software testing. His research interests include Software design and Software testing.

Angelo Susi is a research scientist in the Software Engineering group at Fondazione Bruno Kessler in Trento, Italy. His research interests are in the areas of requirements engineering, goal-oriented software engineering, formal methods for requirements validation, and search-based software engineering. He published more than 90 refereed papers in journals and international conferences such as TSE, TOSEM, IST, SoSyM, FSE, ICSE, RE. He participated in the organization committee of several conferences, such as SSBSE'12 (General Chair), RE'11 (Local and Financial chair) and in program committees of international conferences and workshops (such as AAMAS, ICSOC, CAiSE and SSBSE). He also served as reviewer for several Journals such as TSE, REJ, IST, JSS. He is the scientific manager of the EU FP7 project RISCOSS.

Giuseppe Scanniello received his Laurea and Ph.D. degrees, both in Computer Science, from the University of Salerno, Italy, in 2001 and 2003, respectively. In 2006, he joined, as an Assistant Professor, the Department of Mathematics and Computer Science at the University of Basilicata, Potenza, Italy. In 2015, he became an Associate Professore at the same university. His research interests include requirements engineering, empirical software engineering, reverse engineering, reengineering, software visualization, workflow automation, migration, wrapping, integration, testing, green software engineering, global software engineering, cooperative supports for software engineering, visual languages and e-learning. He has published more than 140 referred papers in journals, books, and conference proceedings. He serves on the organizing of major international conferences and workshops in the field of software engineering. Giuseppe Scanniello also participated in the program committees of several international conferences and workshops and served as reviewer for several primary international journals. He is a member of IEEE and IEEE Computer Society. Giuseppe Scanniello leads both the group and the laboratory of software engineering at the University of Basilicata.

APPENDIX

In this Appendix, we present an instantiation of the fault injection process described in Section 7.3. In particular, let us consider a fault having BCEL-172 as identification number of bug tracker of CommonsBcel. Figure 5 shows how this fault appeared in the online bug tracker system. This fault aroused an `ArrayOutOfBoundsException` when the search functionality is executed with a given input.


https://issues.apache.org
[BCEL-172] Searching bug -...
File Edit View Favorites Tools Help
Agile Board
Export

Commons BCEL / BCEL-172

Searching bug

Type:	Bug	Status:	RESOLVED
Priority:	Major	Resolution:	Duplicate
Affects Version/s:	5.2	Fix Version/s:	6.0
Component/s:	Main		
Labels:	None		

Assignee: Unassigned

Reporter: AK

Votes: 0 Vote for this issue

Watchers: 1 Start watching this issue

Description

It turned out that finder.search methods generates incorrect output.
For example:

```
finder.search("invokespecial")
```

Will find not only some INVOKESPECIAL opcodes but will return InstructionHandle arrays in form:

```
[   invokespecial,    nextOpCode]
```

So instead of returning x matching opcode(s) it returns x+1 matching opcodes in one IntructionHandle array. This generates problem when invoking `finder.search ("invokespecial return")` which will throw

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException
```

Dates

Created: 18/Oct/13 12:38

Updated: 24/Apr/14 15:18

Resolved: 24/Apr/14 15:18

Fig. 5. Screenshot of fault having id = BCEL-172 in the online bug-tracker of CommonsBcel

```
Pattern regex = Pattern.compile(search);
List matches = new ArrayList();
Matcher matcher = regex.matcher(il_string);
while (start < il_string.length() && matcher.find(start)) {
    int startExpr = matcher.start();
    int endExpr = matcher.end();
    int lenExpr = (endExpr - startExpr) + 1;
    InstructionHandle[] match = getMatch(startExpr, lenExpr);
    if ((constraint == null) || constraint.checkCode(match)) {
        matches.add(match);
    }
    start = endExpr;
}
return matches.iterator();
```

(a)

```
Pattern regex = Pattern.compile(search);
List<InstructionHandle[]> matches = new ArrayList<InstructionHandle
Matcher matcher = regex.matcher(il_string);
while (start < il_string.length() && matcher.find(start)) {
    int startExpr = matcher.start();
    int endExpr = matcher.end();
    int lenExpr = (endExpr - startExpr);
    InstructionHandle[] match = getMatch(startExpr, lenExpr);
    if ((constraint == null) || constraint.checkCode(match)) {
        matches.add(match);
    }
    start = endExpr;
}
return matches.iterator();
```

(b)

Fig. 6. (a) code with fault (b) code without fault

From the analysis of fault report, we can observe that the chosen fault affected version 5.2 of CommonsBcel and it was fixed in version 6.0 (RC1). Hence, by looking at posted patch and also at code of CommonsBcel version 5.2 (see Figure 6(a)) and version 6.0_RC1 (see Figure 6(b), we identified where the fault was present in the code, thus understanding how and where to inject it to get a faulty version of CommonsBcel to be used in our experiment. Once that fault was injected, we verify capability of our test suite in detecting it. That is, if at least one test case failed, we choose fault BCEL-172.

The fault BCEL-172 was considered severe because its priority and severity were high and because that fault completely compromised CommonsBcel behavior. Chosen fault was also related to a relevant requirement. In fact, it compromised a functionality critical for the user of CommonsBcel library. The application documentation[5] lists search functionality as one of the key provided functionality and several users identified the faults and reported it in the application bug-tracker (e.g., BCEL-172, 85, 114, 125).

5. http://commons.apache.org/proper/commons-bcel/manual.html