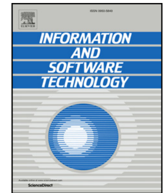
Information and Software Technology 99 (2018) 62–80

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



SPIRITuS: a SimPle Information Retrieval regressIon Test Selection approach

Simone Romano[*,a], Giuseppe Scanniello[a], Giuliano Antoniol[b], Alessandro Marchetto[c]

[a] University of Basilicata, Potenza, Italy
[b] Ecole Polytech. de Montreal, Montreal, QC, Canada
[c] Independent Researcher, Trento, Italy

ARTICLE INFO

Keywords:
SPIRITuS
Regression test case selection
Regression testing

ABSTRACT

*Context:*Regression Test case Selection (RTS) approaches aim at selecting only those test cases of a test suite that exercise changed parts of the System Under Test (SUT) or parts affected by changes.

*Objective:*We present SPIRITuS (SimPle Information Retrieval regressIon Test Selection approach). It uses method code coverage information and a Vector Space Model to select test cases to be run. In a nutshell, the extent of a lexical modification to a method is used to decide if a test case has to be selected. The main design goals of SPIRITuS are to be: *(i)* easy to adapt to different programming languages and *(ii)* tunable via an easy to understand threshold.

*Method:*To assess SPIRITuS, we conducted a large experiment on 389 faulty versions of 14 open-source programs implemented in Java. We were mainly interested in investigating the tradeoff between the number of selected test cases from the original test suite and fault detection effectiveness. We also compared SPIRITuS against well-known RTS approaches.

*Results:*SPIRITuS selects a number of test cases significantly smaller than the number of test cases the other approaches select at the price of a slight reduction in fault detection capability.

*Conclusions:*SPIRITuS can be considered a viable competitor of existing test case selection approaches especially when the average number of test cases covering a modified method increases (such information can be easily derived before test case selection takes place).

1. Introduction

Regression testing is conducted after changes are made to a system in order to ensure that modifications did not alter the expected behavior. The simplest regression test strategy, named *Retest-all*, consists in re-executing the entire test suite of the System Under Test (SUT) on the modified SUT version. As a system evolves, its test suite tends to grow in size. Therefore, Retest-all might not be a viable option because it might require too much time and/or too many resources [1,2]. To deal with this problem, a number of approaches have been proposed. Existing approaches can be classified into three main classes: *(i)* test suite minimization (or test suite reduction); *(ii)* Regression Test case Selection (RTS); and *(iii)* test case prioritization (*e.g.,* [3–10]). Both test suite minimization and RTS approaches seek to reduce the size of the entire test suite preserving its capability to reveal faults. While test suite minimization approaches delete obsolete or redundant test cases from a test suite [11], RTS approaches select a subset of test cases to execute only those that exercise changed parts or parts affected by changes of a SUT [12]. That is, RTS approaches temporarily select test cases (they do not remove selected test cases from test suites). Finally, test case prioritization approaches sort test cases according to one or more criteria [1,13]. Common to the three classes of approaches mentioned before is the assumption of the availability of some a-priori knowledge about the SUT and its test suite (*e.g.,* coverage matrix or test case past fault revealing ability).

RTS approaches are appealing because they allow executing a subset of the entire test suite without removing any test case. However, as underlined by Engström et al. [14], RTS approaches need to be tailored to specific situations, development organizations, requirements, complexity of the problem and preconditions in software applications since no general solution exists. This in turn leads to the development of a number of specialized RTS strategies [12]. They differ from one another according to how they define and identify modifications in a SUT (*i.e.,* when passing from its previous version to the current one), and how they associate test cases to modifications (*i.e.,* how test cases are selected).

In this paper, we present SPIRITuS (SimPle Information Retrieval regressIon Test Selection approach), an RTS approach. To select test

* Corresponding author.
E-mail addresses: simone.romano@unibas.it (S. Romano), giuseppe.scanniello@unibas.it (G. Scanniello), antoniol@ieee.org (G. Antoniol), alex.marchetto@gmail.com (A. Marchetto).

https://doi.org/10.1016/j.infsof.2018.03.004
Received 23 June 2017; Received in revised form 24 January 2018; Accepted 3 March 2018
Available online 27 March 2018
0950-5849/ © 2018 Elsevier B.V. All rights reserved.

cases, SPIRITuS uses method coverage information (*i.e.,* a method coverage matrix) and a Vector Space Model (VSM). In more detail, let P and P' be the previous and current versions of a SUT and let T be the test suite of P. To decide if a method $m' \in P'$ has to be tested or not, SPIRITuS compares the new method version with the previous one (*i.e.,* $m \in P$) via VSM. If the lexical similarity between m' and m is below a given threshold, the method m' is tagged as to be tested and all the test cases exercising m are selected from T (*i.e.,* from the test suite of P) to create a test suite T' with which to test P'. SPIRITuS is tunable: by changing the threshold it is possible to select a lower (or higher) number of test cases. However, it pays to be cautious as reducing the number of test cases may have a price in fault detection capability. Intuitively, the lower the threshold, the fewer test cases will be selected. On the other hand, the higher the threshold, the closer T' will be to T.

To empirically evaluate the SPIRITuS performance (*i.e.,* tradeoff between size and fault detection effectiveness of T'), we compared it against *Diff, Random-75*, and *Ekstazi* [14–16]. The Diff approach executes the test cases that cover methods that are changed. A method m' is considered changed with respect to m, if there is any textual difference to the source code of these methods. Differences are computed via the UNIX `diff` tool. Random-75 randomly selects 75% of test cases in T. Ekstazi tracks dynamic dependencies of test cases on files, then it selects a test case to be run if its dependent files are changed. Ekstazi represents the state of the art for the RTS approaches [17] and this was why we chose it as one of the baseline approaches. SPIRITuS and the baseline RTS approaches were compared on 389 faulty versions of 14 Java programs.

Paper structure. In Section 2, we highlight background and related work, whereas we introduce our approach in Section 3. The design of our experiment is shown in Section 4. The obtained results and their discussion are presented in Section 5, while threats to validly are discussed in Section 6. Final remarks conclude the paper.

2. Background and related work

In this section, we first discuss the RTS problem and the fault seeding. Finally, we summarize RTS approaches and studies on these approaches.

2.1. Regression test case selection

RTS approaches aim at selecting only those test cases that exercise changed parts of the SUT or parts affected by changes to reduce the testing cost, while preserving the same fault detection capability as the original test suite [12]. More formally, the RTS problem is defined as follows [1,2,18]:

- Given a previous version of the SUT, P, the current version of the SUT, P', and a test suite T for P, select a subset of T, T', with which to test P'.

These notations will be used in the rest of the paper. According to the definition above, Retest-all consists in applying T on P'. It is worth mentioning that the evolution of T (*e.g.,* adding a test case to T) is out of the scope of RTS approaches [18] as well as the cost to fix the faults that T' detects or not.

2.2. Seeded faults in empirical investigations

Experiments in regression testing, often, require a set of programs with known faults [19]. One problem is that real programs of realistic size with real faults are difficult to find [19]. To deal with this issue, one common practice is to seed faults, either manually or automatically by applying mutation operators (see the definition in Table 1) into existing programs [20]. A study by Andrews et al. [19] showed that faults seeded by applying mutation operators (*i.e.,* mutation faults, see the definition in Table 1) can be representative of real faults, while hand-seeded faults seem to be harder to detect than real ones. Do and Rothermel [21] studied the use of mutation operators in empirical assessments of test case prioritization techniques. Results indicated that mutation faults can replace real or hand-seeded faults. Do and Rothermel [21] also proposed an experimental procedure largely adopted in the context of regression testing. For example, Do et al. [22] and Hao et al. [23] applied this procedure to evaluate prioritization approaches, while Mirarab et al. [8] to assess a size-constrained RTS approach. Recently, this experimental procedure has been implemented in SMUG [24]. It is a tool prototype (*i.e.,* an Eclipse plug-in) that seeds faults in Java source code. We used this tool in the instrumentation of the experiment conducted to assess SPIRITuS (see Section 4.6). SMUG first compares the previous version of a program, P, and the current one, P', by identifying methods that are added to P' or modified in P'. Then, it creates mutants that involve only these modified/added methods. The created mutants are then used to generate faulty versions of P' with known faults. To this end, SMUG implements the following steps:

Table 1
Definitions and useful concepts.

Mutation operator	It is an operator that, on the basis of a transformation rule, applies a syntactic change to a code region (*e.g.,* a statement or a method). Mutation operators can be applied on either source code or byte-code of a program.
Mutation fault	It is the result of the application of a mutation operator (*i.e.,* the changed code region).
Mutant	It is a version of the program with a mutation fault.
Killed mutant	A mutant is killed if and only if it exists at least one test case in the test suite that detects its mutation fault.
Mutant pool	It is a set of killed mutants.
Mutant group	It a subset of the mutation pool.
Faulty version	It is a version of the program built on the basis of a mutation group. A faulty version contains a (mutation) fault for each mutant in the mutant group.

1. *Identifying modified/added methods.* P and P' are compared to identify methods that are added to P' or modified in P'. To this end, SMUG compares each method $m' \in C'$ to its counterpart $m \in C$, where C' and C are classes with the same fully qualified name in P' and P, respectively. In addition, m' and m must have the same signature (*i.e.,* name and parameter types). If m' and m have at least one changed statement (*i.e.,* whitespace characters and comments are disregarded), then we consider m' a modification of m. If m' does not have a counterpart m, then m' is considered as added to P'. Operations like moving a method from a class to another one, renaming a method, and changing parameter types, are seen as a method added to P' [21].
2. *Creating a mutant pool.* Mutation operators are applied to each modified/added method to create a mutant pool (see the definition in Table 1) for P'. The mutation operators implemented in SMUG are: *Arithmetic OPerator change* (AOP), it replaces an arithmetic operator with another arithmetic operator (*e.g.,* `+` with each of the following operators `-`, `*`, or `/`); *Logical Connector Change* (LCC), it replaces a logical connector with another logical connector (*e.g.,* `&&` with `||`) and a bitwise operator with another bitwise operator (*e.g.,* `&`, `|`, `^`); *Relational Operator Change* (ROC), it replaces a relational operator with another relational operator (*e.g.,* `>` with each of the following operators `<`, `<=`, `>=`, `==`, or `!=`); *Overriding Method Deletion* (OMD), it deletes a declaration of an overriding method; and *Argument Order Change* (AOC), it changes the order of the arguments in a method invocation if there are at least two arguments with compatible types.
3. *Creating mutant groups.* Given a mutant pool, n mutant groups (see the definition in Table 1) are created according to the following criteria: *(i)* each mutant group M_i contains a random number of

killed mutants (see the definition in Table 1), k_i, ranging in between 1 and l; *(ii)* each killed mutant in a mutant group is randomly selected from the mutant pool; *(iii)* each mutant group does not contain mutants whose mutation faults involve the same instruction; and *(iv)* no couple of mutant groups contains the same mutant. The researcher chooses the values for n and l.
4. *Generating faulty versions.* For each mutant group M_i, a faulty version (see the definition in Table 1) of P' with k_i mutation faults is generated. The total number of faulty versions of P' is equal to n. For each seeded fault, we have its position in the source code and the test case/s that detect/s this fault. This information is of primary importance to assess RTS approaches [19].

2.3. Vector space model

The representation of a set of documents (*i.e.,* the corpus) as vectors in a common vector space is known as the Vector Space Model (VSM) [25,26]. It can be used in a number of information retrieval operations ranging from: scoring the textual similarity among documents in the corpus, scoring the textual similarity of documents on a query, document classification, and document clustering. In a VSM, a document d is represented as a vector of numbers, called document vector, so that a point is associated to d. That is, each document is associated with a real valued vector that spans the space of terms in the corpus. Also queries are represented as vectors. If a term t occurs in the document d, its value in the vector is non-zero. The values associated to the terms are known as (term) weights. The simplest approach to compute term weights consists in assigning to each term t a weight equal to the number of its occurrences in the document d. This weighting schema is named as *term frequency* (*i.e.,* $tf_{t,d}$). The dimensionality of document vectors is the number of terms in the corpus. Since VSM does not use a predefined vocabulary or grammar, it can be easily applied to any kind of corpora. The term frequency weighting schema suffers from a critical problem: all terms are considered equally important. In fact, certain terms have little or no discriminating power in determining relevance when executing a query. For instance, cards of books in a library are likely to have the terms *author* and *title*. To deal with this issue, the largest used weighting schema is the *term frequency-inverse document frequency*:

$$tf-idf_{t,d} = tf_{t,d} * idf_t \tag{1}$$

where idf is the inverse document frequency and is computed as follows:

$$idf_t = \log \frac{N}{df_t} \tag{2}$$

N is the number of documents in the corpus, while df_t is the document frequency defined to be the number of documents in the corpus that contain the term t. From a practical perspective, the $tf-idf$ weighting schema assigns higher values either to terms with a high number of occurrences in the document or to terms that appear in a small number of documents.

For a document d, any weighting function that maps the number of occurrences of t in d (included the tf weights above) may be viewed as a quantitative digest of that document. This view of a document is known in the literature as the bag of words model. In this model the exact ordering of the terms in a document is ignored, while the number of occurrences of each term is retained (*e.g.,* the document "Mary is faster than Carmen" is equal to the document "Carmen is faster than Mary").

Since not all the terms are equally important, a different set of techniques can be used to normalize the corpus [25] before the indexing (*i.e.,* the application of VSM): stop word removal, special token elimination, etc. Removed terms, therefore, do not contribute in any way to retrieval and scoring. Other normalization techniques can be applied on the corpus, *e.g.,* splitting identifiers and stemming. Also, these techniques have to be applied before the indexing takes place.

A few approaches have been proposed to quantify the (lexical) similarity between two documents in VSM. Some of these consider the magnitude of the vector difference between two document vectors (Euclidean distance). A drawback of this approach is that two documents with a very similar content can have a significant vector difference since their lengths are very different. To deal with this drawback the standard way of computing the similarity of the documents d_1 and d_2 consists in computing the *cosine similarity* between the corresponding document vectors, $\vec{V}(d_1)$ and $\vec{V}(d_2)$, as follows:

$$SIM(d_1, d_2) = \frac{\vec{V}(d_1)\cdot\vec{V}(d_2)}{|\vec{V}(d_1)||\vec{V}(d_2)|} \tag{3}$$

where the numerator represents the dot product (also known as the inner product) of the two vectors. The denominator is the product of the Euclidean lengths of the vectors. *SIM* assumes values in between 0 and 1, where 1 means that the two documents are identical from the textual point of view. Values below 1 indicate that the two documents are different.

As far as the textual similarity of documents on a query is concerned, VSM can be easily applied by considering a query q as a vector. The query vector is built as the document vectors. To compute the similarity between the query q and a given document d, it suffices to compute the cosine similarity (Eq. (3)) between the vectorial representation of q and d, that is, between $\vec{V}(q)$ and $\vec{V}(d)$, respectively.

VSM is simple (*i.e.,* it does not need specific parameter settings like other text retrieval techniques, such as Latent Dirichlet Allocation — LDA — and Latent Semantic Indexing — LSI), very efficient, and indexing can be easily done in an incremental fashion [25].

2.4. Related work

Rothermel and Harrold [12] identified three main families for code-base RTS approaches: *(i)* coverage-based (they select test cases that exercise changed part of the SUT); *(ii)* minimization-based (they select the smallest subset of test cases that can satisfy some minimum coverage criteria for the modified parts of the SUT); and *(iii)* safe (they guarantee, by construction, that all test cases that can reveal faults are selected).

White and Leung [27] proposed a coverage-based RTS approach that isolates dependencies to the changed parts of a SUT and then selects a subset of T covering the parts within the firewall. More recently, Soetens et al. [28] proposed ChEOPSJ, a firewall-based approach and made available a tool prototype [29]. ChEOPSJ was validated by using mutation testing.

Coverage- and minimization-based RTS approaches work in a similar fashion. The most remarkable difference is that minimization-based approaches seek to select a minimal subset of T that exercises changed parts of the SUT or parts affected by changes. The approach proposed by Fischer et al. [30] falls into this family. This approach uses a 0-1 integer programming algorithm to identify a subset of T that includes at least one test case that exercises a modified part of the SUT. Mirarab et al. [8] addressed a variation of the traditional RTS problem, called size-constrained RTS. They apply an Integer Linear Programming (IP) problem using two different coverage-based criteria. Constraint relaxation is used to find many close-to-optimal solution points and then combined to obtain a final solution (using a voting mechanism). Selected test cases are then prioritized using a greedy algorithm that maximizes coverage in an iterative manner. The authors conducted an empirical evaluation on five programs by using mutation testing. A practical question is that the approach needs to know (or estimate) the number of test cases to be selected in advance.

Safe RTS approaches guarantee by construction that all test cases that can reveal faults are selected. Examples of safe approaches are those proposed by Rothermel and Harrold [31], Vokolos and Frank [15], and Gligoric et al. [16,32]. Rothermel and Harrold's

approach build a graph-based representation (*i.e.,* control-flow-graph) of *P* and *P′*. This representation together with code coverage information are used to detect changes and then decide which test cases have to be selected. Vokolos and Frank's approach, called Pythia, identifies changes by means of textual differences (*i.e.,* `diff`) between *P* and *P′*, then selects those test cases that cover the identified changes. Pythia and Diff (one of the baseline approaches) can be considered as the same approach. The only difference is that Pythia works at file level on C programs, while Diff works at method level on Java programs. Due to their nature, these approaches can work on programs written in any programming language. However, their application on programs written in object-oriented languages (*e.g.,* Java) could turn them into unsafe RTS approaches. Pythia/Diff and SPIRITuS work similarly because they both leverage textual differences and code coverage information. Pythia/Diff uses a simple textual difference, while SPIRITuS uses a VSM to detect changes. According to documented results on the use of information retrieval techniques (*e.g.,* VSM) to code-based analysis (*e.g.,* [33]), we expect that SPIRITuS could provide a more fine-grained analysis of code changes and can reduce the size of *T* more than Diff. Gligoric et al. [32] proposed an RTS approach that relies on distributed version-control systems. More recently, Gligoric et al. [16] proposed Ekstazi (one of the baseline approaches), a firewall-based approach that tracks dynamic dependencies of test cases on files. Ekstazi does not explicitly compare *P* and *P′* but for each test case, it identifies what files it depends on. Executable code such as Java classes as well as external resources such as configuration files are files considered by Ekstazi. A test case is then selected if at least one of its dependent files changed. Similar to Ekstazi, SPIRITuS works with code coverage information. However, while Ekstazi considers all file changes, SPIRITuS compares *P* and *P′* by adopting VSM for identifying "relevant" code changes. Thanks to this comparison between *P* and *P′*, we expect that SPIRITuS could provide a more fine-grained and tunable (by means of the threshold of similarity between *P* and *P′*) analysis of the code changes, thus allowing a better reduction of the size of *T* than Ekstazi. RTS approaches can be safe only under determined conditions when dealing with software written in object-oriented programming languages. This is due to the peculiarities of these kinds of programming languages (*e.g.,* late binding and multithreading), see [34] for more details. Approaches that work at class or module level (*e.g.,* firewall-based approaches) may also select test cases that execute a modified part of a SUT, but these test cases cannot be modification-traversing in any way [1]. In other words, these approaches could select a large number of test cases [16]. Since SPIRITuS works at method level, it is less sensitive to the issue mentioned before.

The three families of RTS approaches proposed by Rothermel and Harrold were extended by Graves et al. [35] by adding other two families often used in practice: Retest-all and ad hoc/random. Retest-all entirely selects *T* to test *P′*. Ad hoc/random RTS approaches are used when due to time constrains it is not possible to apply Retest-all and supporting tools implementing other RTS approaches are not available. For example, developers could randomly select a subset of test cases of *T* and then run it on *P′*.

To solve the RTS problem, change impact analysis approaches can be used to determine the effects of source code modifications on test cases. If a test case is affected by at least a modification, then it should be run. For example, Ren et al. [36] proposed an Eclipse plug-in, called Chianti, that works by capturing atomic-level changes between different program versions of a code base. To predict what other areas of this code base might be affected by a change, dependencies are calculated between these atomic-level changes through the use of call graphs. To identify failure-inducing edits, Chianti then selects affected tests and determines a subset of those changes that might induce test failures. The number of affecting changes related to each test failure may still be too large for manual inspection. To reduce the effort needed to inspect affecting changes, Zhang et al. [37] presented FAULTTRACER. This approach adapts spectrum-based fault localization techniques and applies them in tandem with an enhanced change impact analysis that uses extended call graphs.

Differently from the approaches described before, the SPIRITuS overarching goal is to: *(i)* be easy to adapt to different programming languages; *(ii)* use a simple easy to understand model of source code entities; *(iii)* be fast in selecting test-cases, and *(iv)* be user-tunable. Porting SPIRITuS on a new programming languages requires only to locate function/method bodies and then chop these function/method bodies into tokens. SPIRITuS leverages existing text retrieval engines to efficiently store and represent methods as element of a vector space. Although the use of information retrieval is not new in the regression testing field, see for example Saha et al. [38] who addressed the problem of regression test prioritization by reducing it to a standard information retrieval problem, it is the first time that an information retrieval technique (*i.e.,* VSM) is used in an RTS approach. Computing the similarity between two methods in SPIRITuS is linear in the VSM dimension, while indexing the corpus is linear with the number of methods, it does not need to be recomputed at each new system version since it can be done incrementally. Finally, a developer/tester can fine-tune the SPIRITuS threshold to seek a tradeoff between the number of the selected test cases and their fault detection capability. An appropriate choice of the threshold could allow SPIRITuS to behave similarly to a safe RTS approach (see Section 5.3.3).

3. Approach

SPIRITuS assumes that a method coverage matrix has been gathered by running *T* on *P*. In Fig. 1, we show an activity diagram with object flow that describes the underlying process of SPIRITuS. A description of the phases of this process follows:

1. *Corpus creation.* The bodies of the methods in *P* are extracted to build the corpus. In other words, each method becomes a document *d* in the corpus. We discarded source code comments. The rationale of this choice is that source code comments could not be updated when the corresponding source code is changed [39–41].
2. *Corpus normalization.* SPIRITuS normalizes the extracted text by performing tokenization [25], namely chopping a text into tokens. Tokenization is performed in the same way as the Java compiler [42]. For example, the line of code `list.add(element);` is tokenized into: `list`, `.`, `add`, `(`, `element`, `)`, and `;`. Then, SPIRITuS performs a stop-words removal to cleanup text;

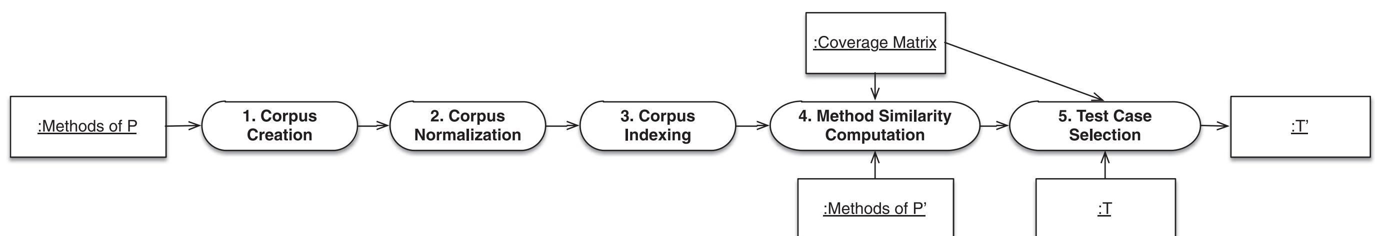


Fig. 1. SPRITuS process.

essentially, full stop, comma, and semicolon are removed. Differently from other text retrieval approaches (*e.g.,* [43,44]), SPIRITuS keeps operators or parentheses [3]. The rationale is that SPIRITuS needs to detect small changes in a method that alter its behavior. For example, removing a pair of parentheses (or a boolean operator) could significantly change the result of an expression and then the behavior of the program.

3. *Corpus indexing*. SPIRITuS models documents (*i.e.,* methods) via a VSM (see Section 2.3) and uses Apache Lucene[1] to index the corpus. In the current SPIRITuS implementation, we use the *term frequency-inverse document frequency* weighting schema [26] to compute the vectorial representations of the methods in the corpus. To quantify the (lexical) similarity between a document in the corpus and a query, SPIRITuS uses the cosine similarity of their vectorial representations.
   VSM has been successfully applied in several software engineering approaches (*e.g.,* [44–46]) and already applied in the context of regression testing, *e.g.,* test case prioritization [3,10]. We opted for VSM because existing research is contradictory on which text retrieval model works best with source code data. For example, Marcus and Maletic [47] observed that LSI [48] performs at least as well as VSM and in some cases LSI outperforms VSM. Differently, Abadi et al. [45] observed that VSM provides better results than LSI and similar results were also obtained by Wang et al. [46]. Other authors advocate for the use of LDA [49]. It is also worth mentioning that the corpus indexing and method similarity computation in SPIRITuS are modular blocks. This is to say that, for example, LSI can be used to replace VSM if needed. We can also note that the use of another text retrieval technique would not alter the general process underlying our approach.
4. *Methods similarity computation*.
   The similarity, *SIM*(*m, m′*), between two methods, *m* (in *P*) and *m′* (in *P′*), is computed by applying the cosine similarity (see Eq. (3)) between the vectorial representations of *m* and *m′*. It is worth mentioning that *m* has to be covered by at least one test case in *T*. As mentioned in Section 2.3, *SIM*(*m, m′*) assumes values in between 0 and 1, where 1 means that the methods *m* and *m′* are equal from the textual point of view. Values below 1 indicate that the two methods are different.
5. *Test case selection.*SPIRITuS compares pair of methods, *m* and *m′*, that *(i)* have the same signature (*i.e., m* has the same name and the same parameter types as *m′*) and *(ii)* both belong to the same class (*i.e., m* belongs to a class with the same fully qualified name as the belonging class of *m′*). Without the assumption above, it is difficult to determine the counterpart of *m, m′*, at a low computational cost. There are three main reasons (besides a simple method removing) why *m* may not have a counterpart *m′*: *(i) m* has been renamed, *(ii) m* has been moved from a class to another, and *(iii)* the parameter types of *m* are changed. To deal with these issues, detection approaches for refactorings could be applied (*e.g.,* [50]). Unfortunately these approaches are time consuming (a few minutes on small/medium programs [50]). If *m* does not have a counterpart *m′*, we assume that *m′* exists and it has an empty body, namely $SIM(m, m') = 0$; more details are given below.
   Once similarities are computed, the selection is performed as follows:

$$T' = \{t \mid SIM(m, m') \leq st \quad and \quad t \ covers \ m\} \tag{4}$$

   where *st* is the *selection threshold* (or simply threshold) and assumes values in the interval [0, 1]. The higher the value of *st*, the larger is the selection. This is to say that *T′* will contain mostly the same test cases of *T* (*e.g.,* if $st = 1$ then $T' = T$) and then the likelihood that *T′* will discover the same faults as *T* increases (*i.e.,* SPIRITuS behaves similarly to a safe RTS approach). On the other hand, the lower the value of *st*, the smaller the selection is. That is, *T′* will contain less test cases than *T*. Therefore, a developer/tester could choose a higher selection threshold if he/she needs a higher fault detection capability, while a lower selection threshold could be chosen when a developer/tester has a quite limited testing budget.
   It is important to underline that parts of SUT affected by changes are "indirectly" handled in SPIRITuS. For example, let $n \in P$ and $o \in P$ be two methods, where *n* calls *o*, and let $t \in T$ be a test case covering both these methods, we consider *n′* (*i.e.,* the counterpart of *n*) affected by the changes to *o*, if:

$$SIM(n, n') > st \quad and \quad SIM(o, o') \leq st \tag{5}$$

   where *o′* is the counterpart of *o*. According to Eq. (4), when *SIM*(*o, o′*) ≤ *st, o′* is considered as changed and *t* is selected. It is worth noting that if *SIM*(*n, n′*) ≤ *st, n′* is considered as changed whatever is the value of *SIM*(*o, o′*), and thus *t* is selected. In case *o* does not have a counterpart *o′* (*e.g., o* has been deleted), $SIM(o, o') = 0$, thus *n′* is a method affected by the changes to *o* (*e.g.,* the deletion of *o*) and *t* is selected. It could happen that a test case $t \in T$ covers a method *o* (in *P*), whose counterpart *o′* (in *P′*) calls a method *q′* (in *P′*) added when passing from *P* to *P′* (*i.e.,* the counterpart of *q′* is not present in *P*). If *SIM*(*o, o′*) ≤ *st*, SPIRITuS selects *t*. This is, SPIRITuS selects a test cases that exercises a method added between *P* and *P′*.

It is worth mentioning that if we apply SPIRITuS for the first time on the current version of the SUT, *P′*, the execution of the phases from 1 to 5 are needed. However, the phases from 1 to 3 can be performed offline because they concern a previous version of the SUT (see Fig. 1). If a new version of the SUT, *P″*, is available and we are interested in applying SPIRITuS between *P* and *P″*, only the execution of the phases 4 and 5 suffices to select a subset of *T, T″*, with which to test *P″*. This is because the phases from 1 to 3 do not concern a new version of the SUT. As for the phases 4 and 5, SPIRITuS could also be executed incrementally. That is, when a developer saves its changes to a method *m* so obtaining *m′*, SPIRITuS can compute *SIM*(*m, m′*) and than decide which test cases have to be selected on the basis of Eq. (4). This is possible thanks to IDEs, such as Eclipse, that provide functionality to notify the change to a code fragment (*e.g.,* to a method).

4. SPIRITuS empirical assessment

To empirically assess SPIRITuS, we conducted an experiment by following the guidelines and recommendations by Wohlin et al. [51]. For replication purposes, we made available a replication package on the web.[2]

4.1. Definition and context

The main goal of our study is to investigate the tradeoff between the reduction of the size of the selections with respect to the original test suites and the loss in fault-detection capability of these selections. According to this goal, we defined and investigated the following research question:

> RQ. Does SPIRITuS reduce the *size* of the original test suites guaranteeing an *effective* fault detection?

In Fig. 2, we show the conceptual model of our experiment. The bottom part of this model shows the main factor under study and the metrics we used to compare SPIRITuS with state of the art and competing approaches. The considered constructs (*i.e.,* size and effectiveness) are shown on the top (right hand side) of the conceptual model.

[1] https://lucene.apache.org

[2] www2.unibas.it/sromano/SPIRITuS.html

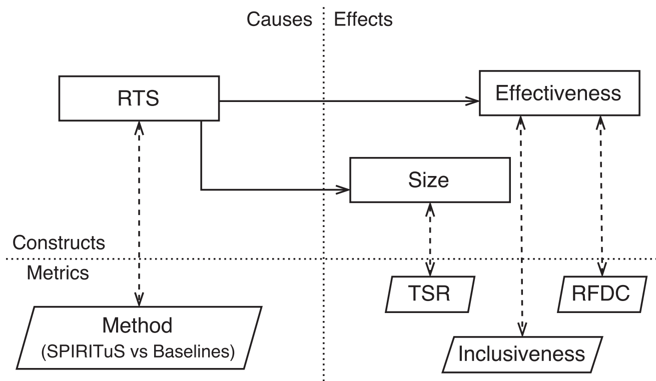


Fig. 2. Conceptual model.

We conducted our experiment on 14 small to medium open-source object-oriented programs (also experimental objects, from here on). We selected these programs because: *(i)* they are written in Java simplifying parsing; *(ii)* they are freely available on the web easing the replications; *(iii)* most of them have been previously used as experimental objects in other empirical studies (*e.g.,* [4]); *(iv)* they belong to different domains; and *(v)* they have non-trivial test suites. In Table 2, we report summary information on the experimental objects. The first column shows the program name, while the second column lists the program versions studied (*i.e., P* and *P′*). The Lines of Code (LOC) are shown in the third column; the number of types (*i.e.,* classes, interfaces, etc.) and methods are reported in the fourth and fifth columns, respectively. The number of commits between the two versions is shown in the sixth column. The size of *T* (*i.e.,* the number of test cases in *T*) is reported in the seventh column, while the eighth column gives the number of faulty versions for each *P′*. The ninth column contains the overall number of mutation faults (summed across all the faulty versions for each *P′*). Finally, a brief description of the experimental objects is given in the last column.

4.2. Planning

We used the experimental procedure proposed by Do and Rothermel [21] (see Section 2.2). This procedure was applied to create 30 faulty versions (*i.e.,* $n = 30$) of *P′*, each of which contained a random number of mutation faults in between 1 and 15 (*i.e.,* $l = 15$). The rationale behind the choice of these values is to have a high number of faulty versions with a typical number of faults [8,21]. In a few cases it was not possible to generate 30 faulty versions of *P′* (*e.g.,* see Spring Context in Table 2) because of the criteria listed in Section 2.2.

Summing up, our empirical assessment can be thought as having evaluated SPIRITuS on a total of 389 pairs of versions, *P* and *P′*, with known faults where each fault could be detected by at least one test case $t \in T$, *i.e.,* it exists in *T* at least one fault-revealing test case *t* for *P′*.

To answer our research question, we compared SPIRITuS with the following RTS approaches:

- *Diff.* It uses method coverage information of *P* and compares *m* and *m′* by means of the UNIX tool `diff`. The method *m′* is different from *m* if `diff` returns that at least one character (in code statements) has been modified, deleted, or added between *m* and *m′*. A test case *t* is selected if it covers a method *m* that results either different from *m′* or removed when passing from *P* to *P′*.
- *Random-75.* It randomly selects 75% of the test cases in *T*.
- *Ekstazi.* It is the approach proposed by Gligoric et al. [16]. It tracks test dependencies on files. For each test case *t*, Ekstazi collects a set of files that are accessed during the execution. A test case *t* is selected by checking if the dependent files are changed. Ekstazi has been conceived to collect dependencies at each revision, but it can also work collecting dependencies at every *n*th revision. When this happens, the cost of frequent collection is avoided, but it might lead to less precise selections [16] (*i.e.,* fault detection test cases could be not selected, see for example the results shown in Section 5.3.1 for Commons IO – one of the used experimental objects).

Diff was chosen because it is based on textual differences, thus it represents a natural choice to compare with SPIRITuS. We chose Random-75 because it is widely used in experiments to assess RTS approaches [14]. Ekstazi represents the state of the art for the RTS approaches [17]. We also considered ChEOPSJ [28], but it did not work on most of the studied experimental objects. This is why we discarded this tool.

4.3. Selected variables

The main factor (or also independent or manipulated variable) is a nominal variable that can assume the following values: SPIRITuS, Diff, Random-75, and Ekstazi. To choose the metrics to quantify our constructs, we exploited the systematic literature review on the assessment of RTS approaches conducted by Engström et al. [14]. According to such a study, we identified and selected the following metrics:

- *Test Suite Reduction* (*TSR*). Let $|T'|$ be the size (*i.e.,* the number of test cases) of *T′*, and let $|T|$ be the size of *T*. *TSR* is computed by the expression $1 - \frac{|T'|}{|T|}$. It assumes a value in between 0 and 1. A high value indicates that the size of the selection *T′* is much lower than the size of the original test suite *T*. Therefore, a high value is desirable. Engström et al. [14] reported that TSR is the most used metric to assess RTS approaches.
- *Inclusiveness*[3] (*I*). Let *M* be the number of fault-revealing test cases in *T′* for *P′*, and *N* be the number of fault-revealing test cases in *T* for *P′*. *I* is given by the expression $\frac{M}{N}$ if $N \neq 0$, 1 if $N = 0$. The desirable value is 1; this means: all fault-revealing test cases are selected.
- *Reduction in Fault Detection Capability*[4] (*RFDC*). Let *M* be the number of faults revealed by *T′* on *P′*, and let *N* be the number of faults revealed by *T* on *P′*. *RFDC* is given by the expression $1 - \frac{M}{N}$ if $N \neq 0$, 1 if $N = 0$. The best value for RFDC is 0.

While TSR concerns the size of the selection (size construct), both I and RFDC are widely used to assess the fault detection effectiveness of *T′* [14] (effectiveness construct). The former effectiveness metric does not take into account that different test cases can reveal the same fault. On the other hand, RFDC takes into account that developers/testers do not need to know all fault-revealing test cases to fix a given fault, but they need to know that at least one test case reveals that fault. We can postulate that the metric RFDC is more practically relevant because provides an indication on the capability of *T′* to reveal faults. For example, let *T* be a test suite with three test cases: t_1, t_2, and t_3. Both t_1 and t_2 reveal the same fault in *P′*, while t_3 does not reveal any fault. If an RTS approach selects only t_1, the I value is 0.5, however the RFDC value is 0.

TSR can be also interpreted as the reduction in term of size achieved by applying an RTS approach with respect to Retest-all, while I and RFDC compare the results of an RTS approach with Retest-all with respect to fault detection effectiveness.

4.4. Hypotheses formulation

We have formulated the following single parametrized null hypothesis:

[3] Also known as test case-related detection effectiveness [14].
[4] Also known as fault-related detection effectiveness [14].

Table 2
Information on the selected programs.

Program	Version	LOC	# Types	# Methods	# Commits	$\|T\|$	# Faulty versions	# Mutation faults	Description
Commons Math (commons.apache.org/proper/commons-math)	3.0 3.1	62,520 77,521	747 912	5506 6615	669	3336	30	200	A library of lightweight, self-contained mathematics and statistics components.
Commons Lang (commons.apache.org/proper/commons-lang)	3.1 3.2	19,499 22,532	150 187	2235 2567	377	2039	30	227	A library of Java utility classes for the classes that are in java.lang's hierarchy.
Commons Configuration (commons.apache.org/proper/commons-configuration)	1.9 1.10	20,773 20,991	189 191	2048 2068	14	1628	27	80	A library to assist in the reading of configuration files in various formats.
Commons IO (commons.apache.org/proper/commons-io)	2.4 2.5	8,839 9,682	109 117	1087 1166	296	869	30	209	A library of utilities to assist with developing IO functionality.
Spring Context (projects.spring.io/spring-framework)	3.1.4 3.2	25,011 25,359	591 613	2974 3011	54	1919	21	101	A module of the Spring Framework. It is a mean to access objects defined and configured.
JFreeChart (sourceforge.net/projects/jfreechart/)	1.0.16 1.0.17	93,320 95,353	626 629	8755 8768	44	2202	30	203	It is a free chart library that supports bar charts, pie charts, line charts, and more.
JGap (jgap.sourceforge.net)	3.5 3.6	28,604 28,943	410 416	3150 3177	82	1387	30	205	It provides basic genetic mechanisms to apply evolutionary principles to problem solutions.
Closure Compiler (developers.google.com/closure/compiler/)	v20160619 v20160713	144,054 144,963	1804 1817	12,577 12,716	92	10,279	30	270	A tool for making JavaScript download and run faster.
Commons BeanUtils (commons.apache.org/proper/commons-beanutils/)	1.8 1.9	11,278 11,535	134 133	1232 1233	208	1061	30	228	It provides easy-to-use wrappers for the classes in the java.lang.reflect and java.beans packages.
Commons Codec (commons.apache.org/proper/commons-codec/)	1.8 1.9	5,781 5,803	84 83	552 553	43	579	21	82	It provides implementations of common encoders and decoders such as Base64, Hex, Phonetic and URLs.
Commons JXPath (commons.apache.org/proper/commons-jxpath/)	1.2 1.3	19,252 18,950	165 179	1657 1692	159	303	30	219	It applies XPath expressions to graphs of objects of all kinds: JavaBeans, DOM etc, including mixtures thereof.
Weka (www.cs.waikato.ac.nz/ml/weka/)	3.6.9 3.6.10	257,981 258,721	2095 2095	18,965 18,973	37	5017	30	254	A collection of machine learning algorithms for data mining tasks.
Commons DBCP (commons.apache.org/proper/commons-dbcp/)	2.0.1 2.1	10,925 11,234	61 62	1383 1409	45	473	30	79	It provides database connection pooling services.
Logback (logback.qos.ch/)	1.0.13 1.1	26,003 26,321	570 579	2990 3033	80	683	20	119	A logging systems.

Table 3
Descriptive statistics and *p*-values obtained by applying the Kruskal-Wallis test. The SPIRITuS threshold is 0.975.

Construct	Metric	Statistic	SPIRITuS	Diff	Random-75	Ekstazi
Size	TSR	Mean	0.351	0.302	0.25	0.158
		Median	0.352	0.307	0.25	0.055
		SD	0.291	0.29	0	0.233
			***p*-value < 0.001**			
Effectiveness	I	Mean	0.904	0.985	0.749	0.953
		Median	1	1	0.75	1
		SD	0.199	0.052	0.029	0.174
			***p*-value < 0.001**			
	RFDC	Mean	0.049	0.015	0.081	0.029
		Median	0	0	0.067	0
		SD	0.128	0.058	0.078	0.122
			***p*-value < 0.001**			

- $\boldsymbol{NH_X}$ - *there is no difference* in the X values (*i.e.,* TSR, I, or RFDC) computed on the selections obtained by applying SPIRITuS, Diff, Random-75, and Ekstazi.

4.5. Data analysis

To test the defined null hypotheses, we used the Kruskal–Wallis test [51]. This is a non-parametric statistical test that allows us to check whether a set of observed independent samples originate from the same distribution, *i.e.,* the null hypothesis is that the medians of samples are equal, while the alternative hypothesis is that at least one sample median among the considered ones is different. If the Kruskal–Wallis test rejects the null hypothesis, the alternative one is accepted. In this latter case, we then applied a post-hoc analysis, *i.e.,* we carried out a pairwise comparison among the results achieved by applying SPIRITuS and by each baseline approach (*e.g.,* "there is not a statistically significant difference between SPIRITuS and Diff with respect to TSR"). To this end, we used a two-sided Mann–Whitney test. This test is non-parametric and checks whether two samples come from the same distribution. We used the Kruskal–Wallis and Mann–Whitney tests because they are well known for their robustness and sensitiveness [51]. For all the statistical test, we decided (as customary) to accept a probability of 5% of committing Type-I-error (*i.e.,* $\alpha = 0.05$). When needed, we applied the Bonferroni correction method [52].

A test of significance checks the presence of a (statistically significant) difference, but it does not provide any information about the magnitude of this difference. To quantify this difference, effect size measures are used in statistics. Reporting effect sizes facilitates the interpretation of the substantive, as opposed to the statistical significance of a result [53]. Having said that, an effect size is a quantitative measure of the strength of a phenomenon/treatment with respect to a baseline. In our data analysis, we used the Cliff's Delta (δ) effect size [54]. The magnitude of the effect size is: negligible if $|\delta| < 0.147$, small if $0.147 \leq |\delta| < 0.33$, medium if $0.33 \leq |\delta| < 0.474$, and large if $|\delta| \geq 0.474$ [55]. In the pairwise comparisons, the sign of δ values is used to understand which approach statistically outperforms the other. For TSR and I, a positive δ value suggests that SPIRITuS outperforms the baseline (*e.g.,* SPIRITuS selects less test cases). As for RFDC, a negative δ value indicates that SPIRITuS outperforms the baseline (*e.g.,* SPIRITuS has a better fault revealing capability).

4.6. Instrumentation

We developed a prototype of a supporting tool implementing the process shown in Section 3. We named this tool as our approach, namely, SPIRITuS. We also developed a tool for Diff and Random-75. The same method-coverage matrices were used to apply Diff and SPIRITuS. As for Ekstazi, we used the Java tool provided by Gligoric et al. [16,56].

We used SMUG [24] to seed faults in the source code in selective fashion (see Section 2.2). In particular, given two subsequent versions of a program, SMUG creates mutants by considering only those methods modified in, or added to, the second version. Therefore, we downloaded both the version P and P' from the web for each experimental object used in our empirical investigation. This prevented the incremental execution of the phases 4 and 5 of SPIRITuS (see Section 3). It is a tradeoff we are willing to take given the extended empirical investigation reported in this paper. According to this design choice the comparison between SPIRITuS and the baselines would be unfair with respect to the time to perform test case selection.

5. Data analysis results and discussion

In this section, we present and discuss the obtained experimental results. We conclude the section presenting results on further analyses and delineating possible practical implications for our research.

5.1. Results

In Table 3, we report descriptive statistics for TSR, I, and RFDC on all the 389 pairs P and P' considering 0.975 as the value for the selection threshold. This value for the threshold was obtained experimentally with the goal to balance the reduction of the size of the selections and their loss in fault-detection capability (see Section 5.3.3). SPIRITuS reduces the number of test cases to run: TSR is 0.351 on average, namely the mean reduction of T' with respect to T is 35.1%. We can see that, on average, SPIRITuS reduces more than the other approaches. Moreover, SPIRITuS ensures a high value of inclusiveness (I is 0.904 on average), at the price of a limited reduction of revealed faults. That is, on average SPIRITuS is not able to reveal 4.9% of the faults T reveals on P' (RFDC average value is 0.049). SPIRITuS overcomes Random-75 (on average, Random-75 is not able to reveal 8.1% of the faults T reveals on P'), but it is slightly worse than Diff and Ekstazi (on average, Diff and Ekstazi are not able to reveal 1.5% and 2.9% of the faults T reveals on P', respectively).

In Fig. 3, we graphically summarize the distribution of the TSR values (on the top), and the distribution of the values for I and RFDC (on the bottom). The overall distribution confirms that SPIRITuS reduces test suites much more than Ekstazi and slightly more than Diff (see Fig. 3a). The boxplot of SPIRITuS seems also more symmetric and less skewed than the others with only exception of Random-75, where the TSR values are always 0.25 because of the definition of this baseline. From Fig. 3b and c, we can also observe that the median values for I are equal to 1 (best value possible) and for RFDC are equal to 0 (best value possible). This holds for all the approaches except for Random-75. Boxplots, however, show the presence of a number of outliers for: SPIRITuS, Diff, and Ekstazi. Such a number of outliers for Diff is less than the other two approaches. This result seems to suggest that SPIRITuS and Ekstazi are more sensitive to the SUT, while Diff less.

In Table 3, we also report the *p*-values obtained by applying the Kruskal–Wallis test. In all the cases, this test allowed us to reject the defined parametric null hypothesis (*i.e.,* the *p*-values are always less than 0.001, that is, the differences are statistically significant by applying the Bonferroni correction). These results justify a post-hoc analysis by performing a pairwise comparison between SPIRITuS and each baseline approach. The obtained results are summarized in Table 4.

For the size construct, the results for TSR suggest that SPIRITuS selects a number of test cases significantly smaller than the number of test cases the other approaches select (see Tables 4). In particular, the Mann–Whitney test allowed us to reject all the defined null hypotheses, *i.e.,* the *p*-values are always less than 0.001. As for Diff and Random-75, the effect size is small (0.159 and 0.219, respectively), while it is medium (0.438) for Ekstazi.

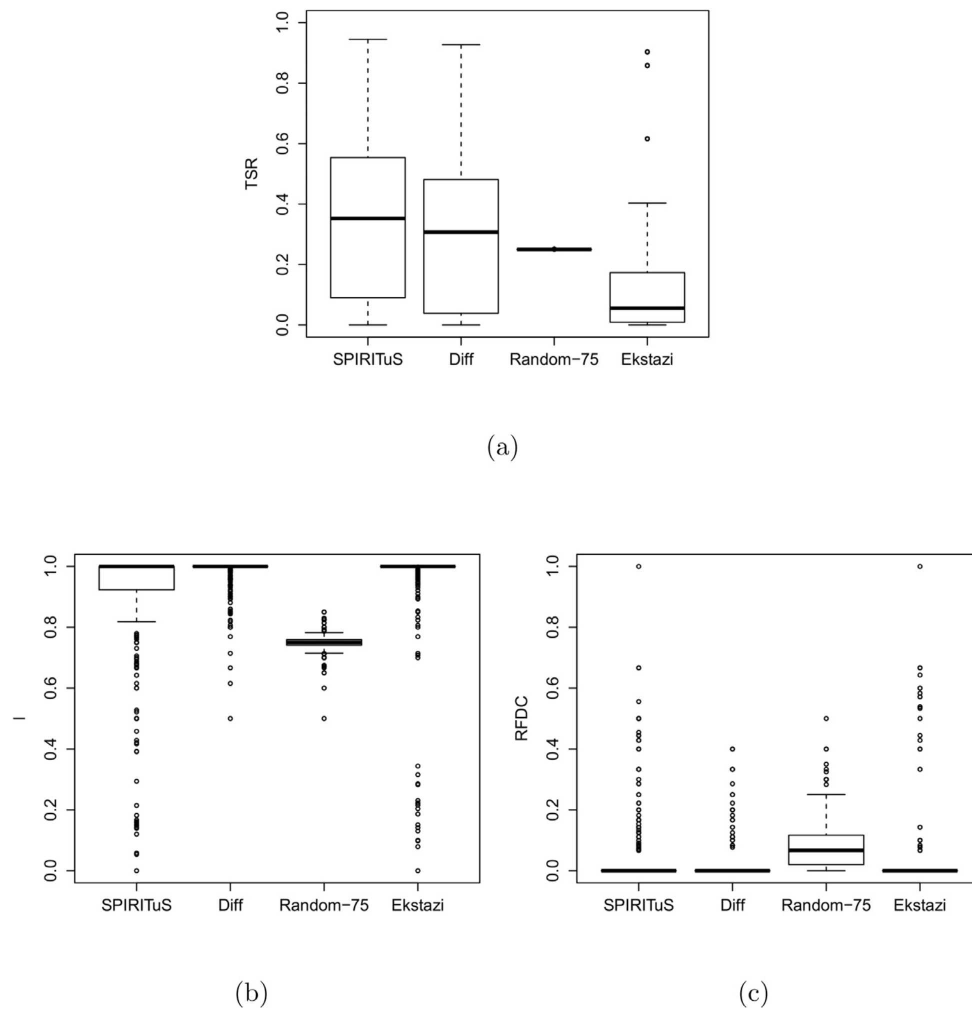


Fig. 3. Boxplots for: (a) TSR, (b) I and (c) RFDC.

As for the effectiveness construct, we can observe that there is a statistically significant difference in all the cases (*p*-values are less than 0.001) between SPIRITUS and the baselines. In particular, Diff and Ekstazi are significantly better than SPIRITuS on I, even if the effect size is small in both the cases (−0.261 and −0.227, respectively). SPIRITuS is significantly better than Random-75 on I and the effect size is large (0.708). As far as RFDC is concerned, Diff and Ekstazi are significantly better than SPIRITuS, but the effect of method is negligible (0.127 and 0.121). SPIRITuS is significantly better than Random-75 on RFDC. The effect size is large (−0.518).

5.2. Discussion

On the basis of the obtained results, we can state that SPIRITuS selects significantly less test cases than the baseline approaches. As for the fault detection effectiveness, SPIRITuS selections detect significantly less faults as compared with Diff and Ekstazi, but the effect size with respect to these approaches is negligible. The results suggest that each test case not selected by SPIRITuS and that, on the contrary, is selected by Diff and Ekstazi reduces marginally the capability of detecting faults. This poses SPIRITuS as a competitor of existing RTS approaches such as Diff and Ekstazi (having different capabilities from them): SPIRITuS has the best selection capability by paying something in terms of fault detection capability. According to developers/testers' needs, SPIRITuS could be adopted rather than other approaches. For instance, a developer/tester could decide to use SPIRITuS since he/she needs the smallest selection of test cases (*e.g.,* in case of a quite limited testing budget), but by being conscious that he/she could loose in terms of fault detection capability. The obtained results also allow comparing SPIRITuS with Retest-all: on average, the SPIRITuS selections are 35.1% smaller than the Retest-all ones; and, on average, the SPIRITuS selections do not detect only 4.9% of faults that Retest-all allows detecting.

Table 4
Pairwise comparisons between SPIRITuS (threshold 0.975) and each baseline with respect to TSR, I, and RFDC. The tested hypotheses are all two-sided because we could not do any postulation on which approach performs better.

Construct	Metric	Diff		Random-75		Ekstazi	
		p-value	Effect size	*p*-value	Effect size	*p*-value	Effect size
Size	TSR	< **0.001**	Small (0.159)	< **0.001**	Small (0.219)	< **0.001**	Medium (0.438)
Effectiveness	I	< **0.001**	Small (−0.261)	< **0.001**	Large (0.708)	< **0.001**	Small (−0.227)
	RFDC	< **0.001**	Negligible (0.127)	< **0.001**	Large (−0.518)	< **0.001**	Negligible (0.121)

Please observe that Retest-all represents the theoretical upper-bound for the fault detection capability, namely any T' cannot detect more faults than T on P'.

Concluding, although our results and the discussion before did not allow us to provide a definitive conclusion about our research question, we can assert that: *SPIRITuS significantly reduces the size of the original test suites with a slight effect on the reduction of detected faults.* That is, SPIRITuS allows obtaining a better tradeoff, as compared with the baseline approaches, between the reduction of the size of the selections and their loss in fault detection capability.

5.3. Further analyses

We conducted four kinds of further analyses: a per-object analysis, a code coverage analysis, a sensitivity analysis, and an analysis of SPIRITuS execution time. The results of these kinds of analysis are presented and discussed in the following subsections.

5.3.1. Per-object analysis

In this further analysis, we considered one experimental object at a time (the used threshold value is 0.975). This was possible because for each program P, we had a number of faulty versions of P' (ranging in between 20 and 30 as shown in Table 2). For each experimental object, we performed a Kruskal–Wallis test. In all the cases, this test returned a p-value less than α. This allowed us to perform a pairwise comparison between SPIRITuS and the baseline approaches on each experimental object. In Table 5, we report descriptive statistics for TSR, I, and RFDC for each experimental object. In Appendix A, we provide the boxplots depicting the values of these metrics grouped by method and experimental object. As for the size construct, SPIRITuS reduces the number of test cases to be run on 13 experimental objects out of 14 (as shown in Table 5, the TSR values range from 0.033 to 0.941 on average). On Commons JXPath, no approach besides Random-75 is capable of reducing the number of test cases to be run. This is due to the high number and impact of changes that lead SPIRITuS, Diff, and Ekstazi to select all the test cases. With respect to the baseline approaches, the mean reduction in terms of test cases is: the best in eight cases out of 14 (the mean TSR values for SPIRITuS range from 0.327 to 0.941); worse only than Random-75 in five cases out of 14 (the mean TSR values for SPIRITuS range in between 0 and 0.148); and worse than Ekstazi in one case out of 14 (the mean TSR values for SPIRITuS and Ekstazi are equal to 0.339 and 0.372 on Weka, respectively).

As for the effectiveness construct, SPIRITuS assures high inclusiveness values and a limited reduction of revealed faults on 13 experimental objects out of 14 (on average, the I values range in between 0.805 and 1, while the RFDC values range in between 0 to 0.097). In particular, SPIRITuS allows detecting the same faults as the original test suite on six experimental objects, while it achieves a low reduction of revealed fault on 7 experimental objects (the mean RFDC values for SPIRITuS range in between 0.006 and 0.097). On JGap, SPIRITuS results are the worst (on average, the I and RFDC values are 0.493 and 0.352, respectively).

In Table 6, we show the results of the two-sided Mann–Whitney tests we performed for each experimental object together with the Cliff's δ effect size values. We also report a symbol (see the outcome column) suggesting which is the RTS approach in each comparison that allowed obtaining the best tradeoff with respect the considered constructs. To this end, we used the comparison matrix shown in Table 7. To define this matrix, we took into account: TSR and RFDC. TSR is the only metric for the size construct, while RFDC can be consider the best for the effectiveness construct because it directly measures the number of faults that T' does not detect with respect to T (see Section 4.3). The headers of both columns and rows are: $+$, $=$, and $-$. The symbol $+$ means that the difference between SPIRITuS and a given baseline is statistically significant in favor of SPIRITuS. For example on TSR, the symbol $+$ means that SPIRITuS reduces a test suite more than a baseline approach and this difference is statistically significant. As for RFDC, the symbol $+$ indicates that SPIRITuS is significantly better than a baseline approach, *i.e.,* the reduction in the fault detection capability of SPIRITuS is significantly less than a baseline approach. On the other hand, the symbol $=$ means that there is not a significant statistically difference between the RTS approaches. The symbol - suggests that the difference between SPIRITuS and a given baseline is statistically significant in favor of the baseline. An entry of the comparison matrix suggests the better approach (if any) in terms of tradeoff between TSR and RFDC. In particular, • and ○ indicate that SPIRITuS is better or worse than a given baseline, respectively. The symbol **X** suggests that the two approaches (*i.e.,* SPIRITuS and the baseline) can be considered comparable. Finally, ◐ and ◐ show two extreme scenarios. The former indicates that SPIRITuS is better on TSR (*i.e.,* there is statistically significant difference), but it is worse on RFDC (*i.e.,* there is statistically significant difference in favor of the baseline approach). On the contrary, ◐ indicates that SPIRITuS is significantly worse on TSR, but it is significantly better on RFDC. In these extreme scenarios, the effect size values would help to understand which approach could be considered better from a practical perspective.

From the results summarized in Table 6, we can note that SPIRITuS is better than the baselines in 24 cases out of 42. SPIRITuS and the baseline are comparable in three cases (**X**). In 10 case (◐) there is a significant effect of SPIRITuS on TSR (with a large effect size) and there is a significant effect of one of the baselines on RFDC. We could speculate that in a real testing scenario, developers/testers could tolerate a slight reduction in fault detection effectiveness with respect to a lower number of test cases to be run. For example, in our study this happens for Weka (SPIRITuS vs Diff), Commons Math (SPIRITuS vs Ekstazi), and LogBack (SPIRITuS vs Ekstazi), where the effect size for RFDC is either small or negligible. In the other cases (◐), there is a significant effect of the baseline on TSR (with a large effect size) and there is a significant effect of SPIRITuS on RFDC (with a large effect size). It is worth mentioning that the latter scenario happens only on Random-75. That is, in such a case, Random-75 reduced more than SPIRITuS at the cost of significantly reducing fault detection effectiveness.

On the basis of the results from this further analysis, we can strength the answer given in Section 5.2 to our research question. In particular, we can state that: even if SPIRITuS does not outperform the baselines on all the experimental objects, it represents a strong competitor for existing RTS approaches since it generally reduces more with a negligible effect on fault detection capabilities. In addition, we can also observe that SPIRITuS generally outperforms baselines when the number of commits between P and P' is less than 100 (see Table 2). SPIRITuS obtained a better tradeoff between TSR and RFDC with respect to Diff and Ekstazi on: Commons Codec, Commons Configuration, Spring Context, Commons DBCP, and Closure Compiler. As for Weka, SPIRITuS outperforms Ekstazi, while there is not a clear winner with respect to Diff (see Table 6). On the contrary, SPIRITuS outperforms Diff on LogBack, while there is not a clear winner with respect to Ekstazi. Although the small number of commits, Diff and Ekstazi outperform SPIRITuS on JGap and JFreeChart with respect to RFDC, while SPIRITuS outperforms these baselines on TSR.

5.3.2. Analyses on code coverage and commits

We also tried to find a pattern in the gathered data to identify in which conditions SPIRITuS outperforms both Diff and Ekstazi in terms of a balance between the size and effectiveness constructs. In the first column of Table 8, we list the experimental objects, while we report the average number of Test cases Covering a modified Method (*i.e.,* TCM) in the second. We observed that SPIRITuS tends to obtain a limited reduction of revealed faults when the average number of test cases covering a modified method increases.

For each experimental object, we also report in Table 8 the number of commits between P and P'. These values were previously shown in Table 2 and are shown again in this section to better support the

Table 5
Descriptive statistics for each experimental object. The SPIRITuS threshold is 0.975.

Program		SPIRITuS			Diff			Random-75			Ekstazi		
		Mean	Median	SD	Mean	Median	SD	Mean	Median	SD	Mean	Median	SD
Commons Math	TSR	0.327	0.327	0.001	0.307	0.307	0	0.25	0.25	0	0.009	0.009	0
	I	0.979	1	0.039	0.999	1	0.005	0.745	0.749	0.03	0.999	1	0.006
	RFDC	0.017	0	0.048	0.009	0	0.039	0.093	0.072	0.088	0	0	0
Commons Lang	TSR	0.372	0.373	0.001	0.324	0.324	0	0.25	0.25	0	0.138	0.143	0.008
	I	0.932	0.997	0.16	0.995	1	0.028	0.747	0.75	0.025	1	1	0
	RFDC	0.07	0	0.116	0	0	0	0.128	0.125	0.059	0	0	0
Commons Configuration	TSR	0.467	0.467	0	0.456	0.456	0	0.25	0.25	0	0.126	0.126	0
	I	0.921	0.986	0.125	0.932	1	0.122	0.751	0.746	0.023	1	1	0
	RFDC	0.006	0	0.032	0.006	0	0.032	0.017	0	0.027	0	0	0
Commons IO	TSR	0.09	0.09	0	0.09	0.09	0	0.25	0.25	0	0.061	0.063	0.004
	I	1	1	0	1	1	0	0.758	0.754	0.026	0.938	1	0.192
	RFDC	0	0	0	0	0	0	0.082	0.081	0.059	0.085	0	0.199
Spring Context	TSR	0.352	0.352	0	0.35	0.35	0	0.25	0.25	0	0.173	0.173	0
	I	0.995	1	0.008	1	1	0	0.747	0.747	0.028	1	1	0.001
	RFDC	0	0	0	0	0	0	0.079	0.075	0.076	0	0	0
JFreeChart	TSR	0.59	0.591	0.001	0.546	0.546	0	0.25	0.25	0	0.388	0.384	0.013
	I	0.888	0.904	0.102	0.929	0.948	0.083	0.739	0.743	0.026	0.929	0.948	0.083
	RFDC	0.097	0.111	0.111	0	0	0	0.149	0.13	0.078	0	0	0
JGap	TSR	0.914	0.915	0.005	0.854	0.854	0	0.25	0.25	0	0	0	0
	I	0.493	0.49	0.307	0.955	0.986	0.076	0.756	0.757	0.019	1	1	0
	RFDC	0.352	0.4	0.175	0.146	0.118	0.139	0.122	0.127	0.057	0	0	0
Closure Compiler	TSR	0.033	0.033	0	0.031	0.031	0	0.25	0.25	0	0.02	0.02	0
	I	1	1	0	1	1	0	0.751	0.75	0.004	1	1	0
	RFDC	0	0	0	0	0	0	0.04	0.038	0.027	0	0	0
Commons BeanUtils	TSR	0.04	0.04	0	0.039	0.039	0	0.25	0.25	0	0.015	0.015	0
	I	1	1	0	1	1	0	0.75	0.752	0.014	0.987	0.998	0.031
	RFDC	0	0	0	0	0	0	0.054	0.041	0.051	0	0	0
Commons Codec	TSR	0.941	0.945	0.007	0.928	0.928	0	0.25	0.25	0	0.886	0.903	0.022
	I	0.847	0.75	0.151	1	1	0	0.754	0.75	0.017	1	1	0
	RFDC	0	0	0	0	0	0	0.009	0	0.018	0	0	0
Commons JXPath	TSR	0	0	0	0	0	0	0.251	0.251	0	0	0	0
	I	1	1	0	1	1	0	0.753	0.751	0.021	1	1	0
	RFDC	0	0	0	0	0	0	0.07	0.05	0.059	0	0	0
Weka	TSR	0.339	0.38	0.087	0.034	0.034	0	0.25	0.25	0	0.372	0.616	0.304
	I	0.873	1	0.208	1	1	0	0.75	0.748	0.019	0.545	0.301	0.406
	RFDC	0.023	0	0.064	0	0	0	0.074	0.072	0.04	0.287	0.367	0.282
Commons DBCP	TSR	0.148	0.148	0	0.074	0.074	0	0.25	0.25	0	0.034	0.034	0
	I	0.805	0.983	0.27	1	1	0	0.732	0.75	0.067	1	1	0
	RFDC	0.044	0	0.185	0	0	0	0.149	0.1	0.135	0	0	0
LogBack	TSR	0.552	0.554	0.008	0.481	0.481	0	0.25	0.25	0	0.235	0.235	0
	I	0.953	1	0.135	0.982	1	0.025	0.756	0.756	0.028	0.995	1	0.009
	RFDC	0.047	0	0.086	0.04	0	0.069	0.029	0.018	0.032	0.003	0	0.015

discussion of the results.

According to the results reported in Tables 6 and 8, we can delineate a number of patterns in the data with respect to TCM and the number of commits. In particular, we can observe that SPIRITuS outperformed both Diff and Ekstazi (see the bottom of Table 8) on the following programs: Closure Compiler, Commons Configuration, Commons DBPC, and Spring Context. For these programs, the TCM values range in between 98 and 1298 and the number of commits ranges in between 14 and 92. The number of commits seems relatively small (as compared with the other programs). Also on Commons BeanUtils, SPIRITuS outperformed both Diff and Ekstazi. The difference here is that the number of commits is higher (*i.e.,* 208). For the programs whose TCM values are less than (or equal to) 41 there is not a clear winner between our approach and the baselines whatever is the number of commits between P and P'. On the basis of these results, we can speculate that when the number of TCM is high and the number of commits is low, SPIRITuS outperforms Diff and Ekstazi. The average number of test cases covering a modified method can be easily computed before applying an RTS approach as well as the number of commits. That is, TCM and the number of commits could represent easy indicators to early have an idea about the performance of SPIRITuS in terms of tradeoff between size and effectiveness of the selections. Moreover, on the basis of this number, developers/testers could decide to increase the selection threshold to reduce TSR and to increase RFDC, or vice-versa decreasing such a threshold.

It is also worth noting that the TCM values for Commons BeanUtils, Commons Math, and Commons JXPath are: 77, 54, and 49, respectively. On the other hand, the number of commits for these programs is: 208, 669, and 159, respectively. Therefore, both the values for TCM and the number of commits seem high as compared with the other programs. In this scenario, SPIRITuS does not clearly outperform both Diff and Ekstazi. On the basis of this result and the pattern highlighted before, we can speculate that TCM is a more important indicator to estimate the performances of SPIRITuS in terms of tradeoff between size and effectiveness of the selections. This latter pattern allows us to speculate that SPIRITuS remains a viable solution in case of TCM values and the number of commits are both high.

5.3.3. Sensitivity analysis

We studied how the SPIRITuS results vary according to the selection thresholds (*i.e., st*, see Section 3). In particular, we computed TSR, I, and RFDC for each program using threshold values that assume values in the interval 0.01 and 0.99, considering an increment of 0.01. The choice of this increment was to have a manageable number of configurations to be analyzed for each program and the corresponding versions. On the basis of the obtained results, we divided our sensitivity analysis into two steps. In the first step, we analyzed the obtained values to find the threshold (common to all the programs) allowing us to

Table 6
Pairwise comparisons between SPIRITuS (threshold 0.975) and baselines on each experimental object. The tested hypotheses are two-sided.

Program		Diff			Random-75			Ekstazi		
		p-value	Effect size	Outcome	*p*-value	Effect size	Outcome	*p*-value	Effect size	Outcome
Commons Math	TSR	< **0.001**	Large (1)	•	< **0.001**	Large (1)	•	< **0.001**	Large (1)	◐
	I	**0.003**	Small (− 0.321)		< **0.001**	Large (1)		**0.002**	Medium (− 0.334)	
	RFDC	0.402	Negligible (0.067)		< **0.001**	Large (− 0.681)		**0.042**	Negligible (0.133)	
Commons Lang	TSR	< **0.001**	Large (1)	◐	< **0.001**	Large (1)	•	< **0.001**	Large (1)	◐
	I	< **0.001**	Medium (− 0.459)		< **0.001**	Large (0.933)		< **0.001**	Large (− 0.5)	
	RFDC	< **0.001**	Medium (0.4)		< **0.001**	Large (− 0.53)		< **0.001**	Medium (0.4)	
Commons Configuration	TSR	< **0.001**	Large (1)	•	< **0.001**	Large (1)	•	< **0.001**	Large (1)	•
	I	0.601	Negligible (− 0.08)		< **0.001**	Large (0.841)		< **0.001**	Large (− 0.556)	
	RFDC	1	0		**0.002**	Medium (− 0.355)		0.336	Negligible (0.037)	
Commons IO	TSR	1	0	X	< **0.001**	Large (− 1)	◐	< **0.001**	Large (1)	•
	I	-	0		< **0.001**	Large (1)		< **0.001**	Medium (0.4)	
	RFDC	-	0		< **0.001**	Large (− 0.8)		< **0.001**	Medium (− 0.367)	
Spring Context	TSR	< **0.001**	Large (1)	•	< **0.001**	Large (1)	•	< **0.001**	Large (1)	•
	I	**0.005**	Medium (− 0.333)		< **0.001**	Large (1)		**0.046**	Small (− 0.261)	
	RFDC	-	0		< **0.001**	Large (− 0.81)		-	0	
JFreeChart	TSR	< **0.001**	Large (1)	◐	< **0.001**	Large (1)	•	< **0.001**	Large (1)	◐
	I	0.093	Small (− 0.249)		< **0.001**	Large (0.788)		0.093	Small (− 0.249)	
	RFDC	< **0.001**	Large (0.6)		**0.015**	Medium (− 0.363)		< **0.001**	Large (0.6)	
JGap	TSR	< **0.001**	Large (1)	◐	< **0.001**	Large (1)	◐	< **0.001**	Large (1)	◐
	I	< **0.001**	Large (− 0.878)		**0.001**	Large (− 0.52)		< **0.001**	Large (− 0.967)	
	RFDC	< **0.001**	Large (0.64)		< **0.001**	Large (0.76)		< **0.001**	Large (0.9)	
Closure Compiler	TSR	< **0.001**	Large (1)	•	< **0.001**	Large (− 1)	◐	< **0.001**	Large (1)	•
	I	0.161	Negligible (− 0.067)		< **0.001**	Large (1)		0.161	Negligible (− 0.067)	
	RFDC	-	0		< **0.001**	Large (− 0.9)		-	0	
Commons BeanUtils	TSR	< **0.001**	Large (1)	•	< **0.001**	Large (− 1)	◐	< **0.001**	Large (1)	•
	I	-	0		< **0.001**	Large (1)		< **0.001**	Large (0.5)	
	RFDC	-	0		< **0.001**	Large (− 0.867)		-	0	
1.7cmCommons Codec	TSR	< **0.001**	Large (0.81)	•	< **0.001**	Large (1)	•	< **0.001**	Large (1)	•
	I	< **0.001**	Large (− 0.524)		0.461	Negligible (0.134)		< **0.001**	Large (− 0.524)	
	RFDC	-	0		**0.020**	Small (− 0.238)		-	0	
Commons JXPath	TSR	-	0	X	< **0.001**	Large (− 1)	◐	-	0	X
	I	-	0		< **0.001**	Large (1)		-	0	
	RFDC	-	0		< **0.001**	Large (− 0.9)		-	0	
Weka	TSR	< **0.001**	Large (1)	◐	< **0.001**	Large (0.867)	•	0.175	Small (− 0.2)	•
	I	< **0.001**	Medium (− 0.467)		< **0.001**	Large (0.492)		**0.011**	Medium (0.364)	
	RFDC	**0.011**	Small (0.2)		< **0.001**	Large (− 0.72)		**0.001**	Medium (− 0.439)	
Commons DBCP	TSR	< **0.001**	Large (1)	•	< **0.001**	Large (− 1)	◐	< **0.001**	Large (1)	•
	I	< **0.001**	Large (− 0.633)		**0.022**	Medium (0.344)		< **0.001**	Large (− 0.633)	
	RFDC	0.082	Negligible (0.1)		< **0.001**	Large (− 0.766)		0.082	Negligible (0.1)	
LogBack	TSR	< **0.001**	Large (1)	•	< **0.001**	Large (1)	•	< **0.001**	Large (1)	◐
	I	0.801	Negligible (− 0.045)		< **0.001**	Large (0.9)		0.095	Small (− 0.265)	
	RFDC	0.947	Negligible (0.013)		0.432	Negligible (− 0.135)		**0.032**	Small (0.265)	

" - " means that the distributions for SPIRITuS and the baseline are equal.

Table 7
Comparison matrix.

		TSR		
		+	=	−
RFDC	+	•	•	◐
	=	•	X	○
	−	◐	○	○

Table 8
TCM values and number of commits (# Commits) between P and P' for each experimental object.

Program	TCM	# Commits
Commons Lang	10	377
Commons IO	14	296
Commons Codec	17	43
JGap	25	82
LogBack	36	80
JFreeChart	41	44
Commons JXPath	49	159
Commons Math	54	669
Commons BeanUtils	77	208
Spring Context	98	54
Commons DBCP	155	45
Commons Configuration	173	14
Weka	331	37
Closure Compiler	1298	92

have a balance between the constructs studied in our experiment. As mentioned before, this analysis has been exploited to study RQ1. In the second step, we individually studied the programs and the threshold values with respect to TSR and RFDC. In particular, we focused on each program to find the best threshold value that allows SPIRITuS to lose the smallest number of faults and, at the same time, to maximize TSR. This further analysis has a twofold goal. The first goal is to deepen the sensitiveness of our proposal. The second goal is to compare SPRITuS with Diff and Ekstazi under the following condition: reduction in the size of the original test suite without any loss in fault detection capability or with an unimportant loss in fault detection capability.

As for the first step of our sensitivity analysis, we observed that there were not significant variations in the average values for TSR, I, and RFDC when choosing a selection threshold in between 0.95 and 0.99 (see Fig. 4). It seems that when choosing threshold values in this interval SPIRITuS is not very sensitive with respect to TSR, I, and RFDC. Values for the selection threshold less than 0.95 produced selections

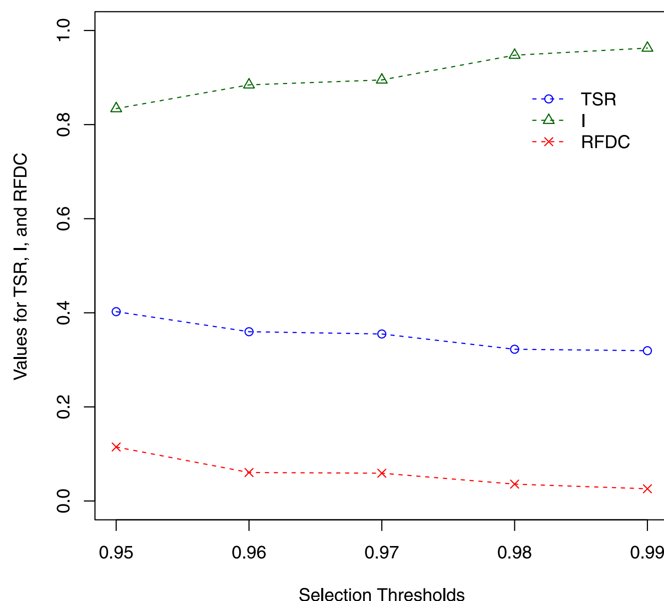


Fig. 4. Line-plot for the SPIRITuS selection thresholds from 0.95 to 0.99. Symbols represent average values for TSR, I, and RFDC, computed on the experimental objects.

much less effective to reveal faults with respect to *T*. Please note that we did not consider 1 because with this selection threshold SPIRITuS behaves like Retest-all. In our experiment, we used 0.975 as selection threshold because this value allowed us to obtain the best results on the used experimental objects with respect to a balance of the metrics used to estimate the considered constructs.

As far as the second step of our sensitivity analysis is concerned, we report in Table 9 some descriptive statistics (*i.e.,* mean, median, and standard deviation) for each program considering the threshold value that allowed SPIRITuS to lose the smallest number of faults and, at the same time, to maximize the reduction of the original test suite. We report the descriptive statistics also for Diff and Ekstazi. These statistics are the same as shown in Table 5. This allowed a faster comparison between SPIRITuS, Diff, and Ekstazi. For each program, we also show the threshold value that produced the highest reduction of the size of the original test suite without any loss in fault detection capability (or with an unimportant loss in fault detection capability). It is worth mentioning that for the greater part of the programs, we found a threshold value that allowed SPIRITuS to behave as a safe RTS approach. In a few cases this was not possible due to the nature of the approach and to the kind of programming language with which the programs were written (*i.e.,* Java). In these cases, the average RFDC values are very close to 0 and the standard deviation is low. It is also worth mentioning that also Diff and Ekstazi selected test suites that were not able to identify the same faults as Retest-all, *i.e.,* the mean RFDC value was greater than 0 in a few cases for both Diff and Ekstazi (in Section 6 possible motivations behind this result are presented).

The results summarized in Table 9 seem to confirm that SPIRITuS is not very sensitive when choosing selection thresholds in between 0.95 and 0.99. In fact, for the greater part of the programs the threshold values in this interval allowed us to have the best results in terms of the size of the selections and loss in fault detection capability (*i.e.,* the RFDC values are either 0 or very close to 0). We obtained threshold values less than 0.95 in four cases: Commons Configuration, Spring Context, Closure Compiler, and Commons JXPath. The RFDC is always equal to 0 except for Commons Configuration (Diff also obtained RFDC values greater than 0). Since the thresholds for these programs are less then 0.975 (*i.e.,* the threshold value used to study RQ1) the TSR values are even greater than those presented and discussed in Section 5.1 and Section 5.2, respectively.

On the basis of the results shown in Table 9, we can also claim that SPIRITuS is more effective than Diff and Ekstazi since it produces smaller selections without any loss in fault detection capability (*i.e.,* in eight cases) or with an unimportant loss in fault detection capability (*i.e.,* six cases). We can then conclude that properly choosing the threshold, SPIRITuS reduces more than Diff and Ekstazi, so decreasing the cost to perform regression testing. In this respect, SPIRITuS reduces up to 74.5% and up to 97% with respect to Diff (*i.e.,* Closure Compiler) and Ekstazi (*i.e.,* Commons Math), respectively. On overage, SPIRITuS reduces 22.5% more than Diff and 66.1% more than Ekstazi.[5]

5.3.4. SPIRITuS execution time

To complete the SPIRITuS assessment, we also gathered the time to apply it on each experimental object (using 0.975 as the threshold value). To this end, we used a PC equipped with 2.50 GHz Intel (quad) Core i7, 16 GB of RAM, and Windows 10 (64-bit) as operating system.

In Table 10, we report the gathered time (expressed in seconds). The shown times are averaged with respect to all the faulty versions of each experimental object. In the second column of Table 10, we report the sum of the (average) times to execute the first three phases of SPIRITuS: (1) Corpus Creation, (2) Corpus Normalization, and (3) Corpus Indexing. We aggregated the time needed to perform these phases because (as mentioned in Section 3) they can be performed together off-line on the previous version of the SUT. In the third column, we report the sum of the (average) times to perform the last two phases: (4) Methods Similarity Computation and (5) Test Case Selection. To compute these phases, SPIRITuS needs both the output of the previous phases and the current version of the SUT. The (average) time to entirely execute the underlying process of SPIRITuS is shown in the fourth column.

The gathered data show the following pattern: the time to perform the phases 4 and 5 is significantly less than the time to perform the phases 1, 2, and 3. For example, the time to perform the phases 1, 2, and 3 is about the 99% of the total cost to apply SPIRITuS. However, there is still room for performance improvements with respect to the implementation of these phases. To this end, a viable solution could consist in integrating SPIRITuS with a version-control system. For example, each time a developer commit a new source file the methods of this file could be added to the corpus.

We did not perform a comparison end-to-end among SPIRITuS and the baseline approaches because of their differences. For example, there is a difference between the underlying processes of SPIRITuS and Ekstazi (*e.g.,* the gathering of the code coverage information is a part of Ekstazi as well as the execution of the selected test cases). In addition a fair comparison is also prevented due to the experimental instrumentation (see Section 4.6).

5.4. Implications and future extensions

We focus on the researcher and the practitioner perspectives for the discussion of the implications and future extensions for our research.

- SPIRITuS could be considered as a competitor of existing test case selection approaches since we observed that different and specific capabilities lead SPIRITuS to achieve specific results: it has shown a non-trivial capability of selecting small subsets of test cases at the price of slightly reducing the fault detection capability of selected test cases. This indicates that the adoption of SPIRITuS rather than existing approaches depends on the developer/tester's needs: *(i)* if he/she focuses on the smallest selection of test cases and a slight reduction in fault detection capability is tolerable, he/she can adopt

[5] We did not consider Weka since Ekstazi selected on average less test cases than SPIRITuS (TSR values: 0.372 vs 0.034) at the cost a sensitive loss in fault detection capability (RFDC values: 0.287 vs 0.)

Table 9
Descriptive statistics for each program considering the threshold value that allowed SPIRITuS to lose the smallest number of faults and, at the same time, to maximize TSR.

Program		SPIRITuS			Diff			Ekstazi		
		Mean	Median	SD	Mean	Median	SD	Mean	Median	SD
Commons Math	Threshold		0.95							
	TSR	0.331	0.331	0	0.307	0.307	0	0.009	0.009	0
	RFDC	0.017	0	0.048	0.009	0	0.039	0	0	0
Commons Lang	Threshold		0.95							
	TSR	0.356	0.356	0.002	0.324	0.324	0	0.138	0.143	0.008
	RFDC	0.021	0	0.065	0	0	0	0	0	0
Commons Configuration	Threshold		0.34							
	TSR	0.506	0.506	0	0.456	0.456	0	0.126	0.126	0
	RFDC	0.006	0	0.032	0.006	0	0	0	0	0
Commons IO	Threshold		0.95							
	TSR	0.131	0.131	0	0.09	0.09	0	0.061	0.063	0.004
	RFDC	0	0	0	0	0	0	0.085	0	0.2
Spring Context	Threshold		0.72							
	TSR	0.391	0.391	0	0.35	0.35	0	0.173	0.173	0
	RFDC	0	0	0	0	0	0	0	0	0
JFreeChart	Threshold		0.98							
	TSR	0.587	0.587	0.000	0.546	0.546	0	0.387	0.384	0.013
	RFDC	0.076	0.08	0.101	0	0	0	0	0	0
JGap	Threshold		0.99							
	TSR	0.894	0.894	0.001	0.854	0.854	0	0	0	0
	RFDC	0.19	0.2	0.145	0.146	0.118	0.139	0	0	0
Closure Compiler	Threshold		0.43							
	TSR	0.123	0.123	0	0.031	0.031	0	0.02	0.02	0
	RFDC	0	0	0	0	0	0	0	0	0
Commons BeanUtils	Threshold		0.98							
	TSR	0.04	0.04	0	0.039	0.039	0	0.015	0.015	0
	RFDC	0	0	0	0	0	0	0	0	0
Commons Codec	Threshold		0.95							
	TSR	0.945	0.945	0	0.928	0.928	0	0.886	0.903	0
	RFDC	0	0	0	0	0	0	0	0	0
Commons JXPath	Threshold		0.68							
	TSR	0.01	0.01	0	0	0	0	0	0	0
	RFDC	0	0	0	0	0	0	0	0	0
Weka	Threshold		0.98							
	TSR	0.034	0.034	0	0.034	0.034	0	0.372	0.616	0.304
	RFDC	0	0	0	0	0	0	0.287	0.367	0.282
Commons DBCP	Threshold		0.98							
	TSR	0.146	0.146	0	0.074	0.074	0	0.034	0.034	0
	RFDC	0	0	0	0	0	0	0	0	0
LogBack	Threshold		0.98							
	TSR	0.52	0.52	0	0.481	0.481	0	0.235	0.235	0
	RFDC	0.04	0	0.069	0.04	0	0.069	0.003	0	0.015

SPIRITuS; and *(ii)* if he/she focuses on having the highest fault detection capability at the price of not-so-small selections, he/she can adopt other RTS approaches.

- If a method m' is obtained from m by only renaming it, SPIRITuS considers m as deleted from P to P'. This could imply that SPIRITuS (but also approaches like Diff) erroneously selects test cases covering m. Enhancing SPIRITuS to properly handle method renaming could reduce $|T'|$. This is clearly relevant for the researcher. Possibly, existing approaches [50,57,58] could be adapted to better deal with renaming issues.
- If the selection threshold is 1 then $T' = T$ and SPIRITuS behaves like Retest-all. On the other hand, threshold values close to 1 allow SPIRITuS to behave like a textual difference approach (*e.g.,* Diff). Clearly, this value of the selection threshold depends on the SUT. We have also experimentally observed that a selection threshold equal to 0.975 allows obtaining a tradeoff between the size of the selection and fault detection effectiveness. In addition, SPIRITuS seems to have a limited sensitivity to the selection threshold because slight variations in the threshold slightly affect SPIRITuS results. Outcomes from our sensitivity analysis (see Section 5.3.3) are clearly relevant for the practitioner, who can choose a value close to our experimental tradeoff and be confident that selection results are

Table 10
Execution time (expressed in seconds) for SPIRITuS phases.

Program	Phases 1–3	Phases 4–5	All phases
Commons Math	11.2493	0.0038	11.2531
Commons Lang	0.5951	0.0015	0.5966
Commons Configuration	0.7647	0.0016	0.7663
Commons IO	0.3354	0.0002	0.3357
Spring Context	1.7962	0.0011	1.7973
JFreeChart	15.7901	0.006	15.7961
JGap	1.6511	0.0016	1.6526
Closure Compiler	17.8341	0.0017	17.8358
Commons BeanUtils	0.4688	0.0002	0.4690
Commons Codec	0.7326	0.0007	0.7333
Commons JXPath	0.7374	0.0006	0.7380
Weka	25.0032	0.0077	25.0109
Commons DBCP	0.3715	0.0009	0.3724
Logback	1.8120	0.0014	1.8134

close to an optimal configuration. This point is also relevant for the researcher, who could be interested in deepening the effect of the selection threshold on the SPIRITuS results.

- The use of SPIRITuS does not require a complete and radical process change within a software company. This outcome can be considered relevant for the practitioner. In fact, 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 [59]. This is why the results of our study could promote the transferring of the developed technology to the software industry. This is of particular interest for the practitioner. The researcher could be interested in identifying opportunities (*e.g.,* industrial case studies and experiments) to speed up this process.
- SPIRITuS has been designed to be easy to adapt to different programming languages. Although our results cannot be generalized to programs written in programming languages different from Java, the researcher could be interested in studying the application of SPIRITuS in the context of programs written in other languages.

6. Threats to validity

To understand strengths and limitations of our experiment, in the following paragraphs we discuss threats that could affect the validity of the results.

- *Construct validity* threats concern the relationship between theory and observation. In our study, construct validity threats are due to: the used mutation operators and metrics. However, the used mutation operators are widely adopted, while the metrics used to quantify our constructs represent the standard to assess RTS approaches [14].
- *Internal validity* threats concern factors internal to our study that could have influenced obtained results. The implementation of the procedure used in our experiment might threaten the validity of the results. Also, the implementations of both SPIRITuS and the baseline approaches (*e.g.,* Ekstazi) could have an unexpected effect on the selections, *e.g.,* these implementations might contain bugs. Finally, the used test cases might also affect the results in an unexpected way. In particular, developers/testers could not have defined these test case to perform regression testing.
- *Conclusion validity* threats concern the relationship between the treatments and the outcomes. In each step of our analysis, we applied the proper statistical tests. Then, when discussing findings we kept into account ranges of acceptability. Also, the chosen baseline approaches might also threaten conclusion validity. However, these approaches can be considered a natural choice to compare with SPIRITuS (see Section 4.2). Finally, the used measures do not provide any indication on the cost to fix undetected and detected faults. That is, unsafe approaches could produce selections whose test cases do not detect faults that are expensive to fix as compared with detected ones. This concern might be relevant for SPIRITuS, but also for all those approaches that do not guarantee that all the fault revealing test cases are selected.
- *Reliability validity* threats concern the possibility of replicating our study. We provide all the details needed to replicate our experiment. We also made available on the web our full replication package comprising experimental objects and raw data.
- *External validity* threats concern the possibility of generalizing our results. Although for our empirical assessment we chose programs previously used in other studies as experimental objects and these programs covers different domains, we cannot guarantee that our findings can be generalized to the universe of Java programs. However, the set of programs used for our empirical assessment can be considered extensive enough as compared with previous studies (*e.g.,* [28,34]).

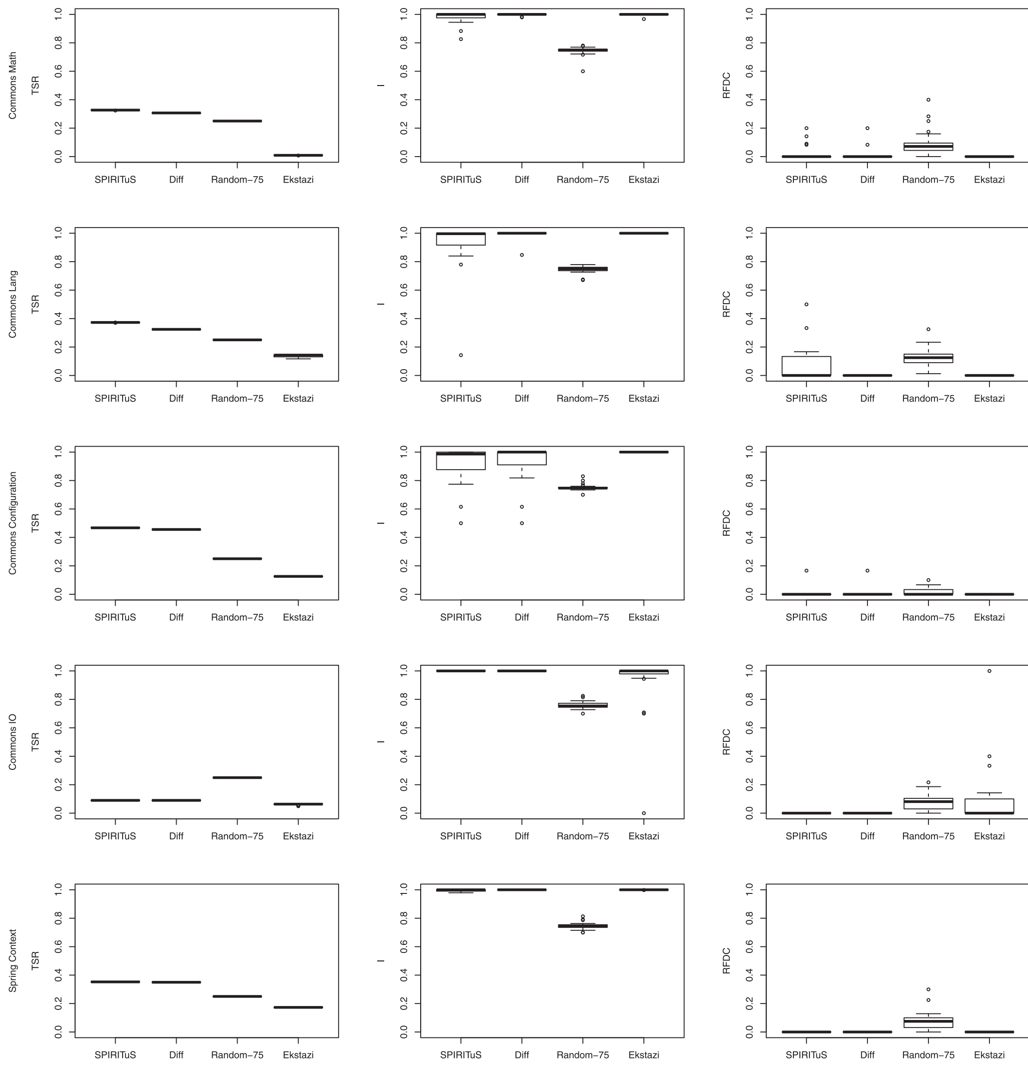
7. Conclusion

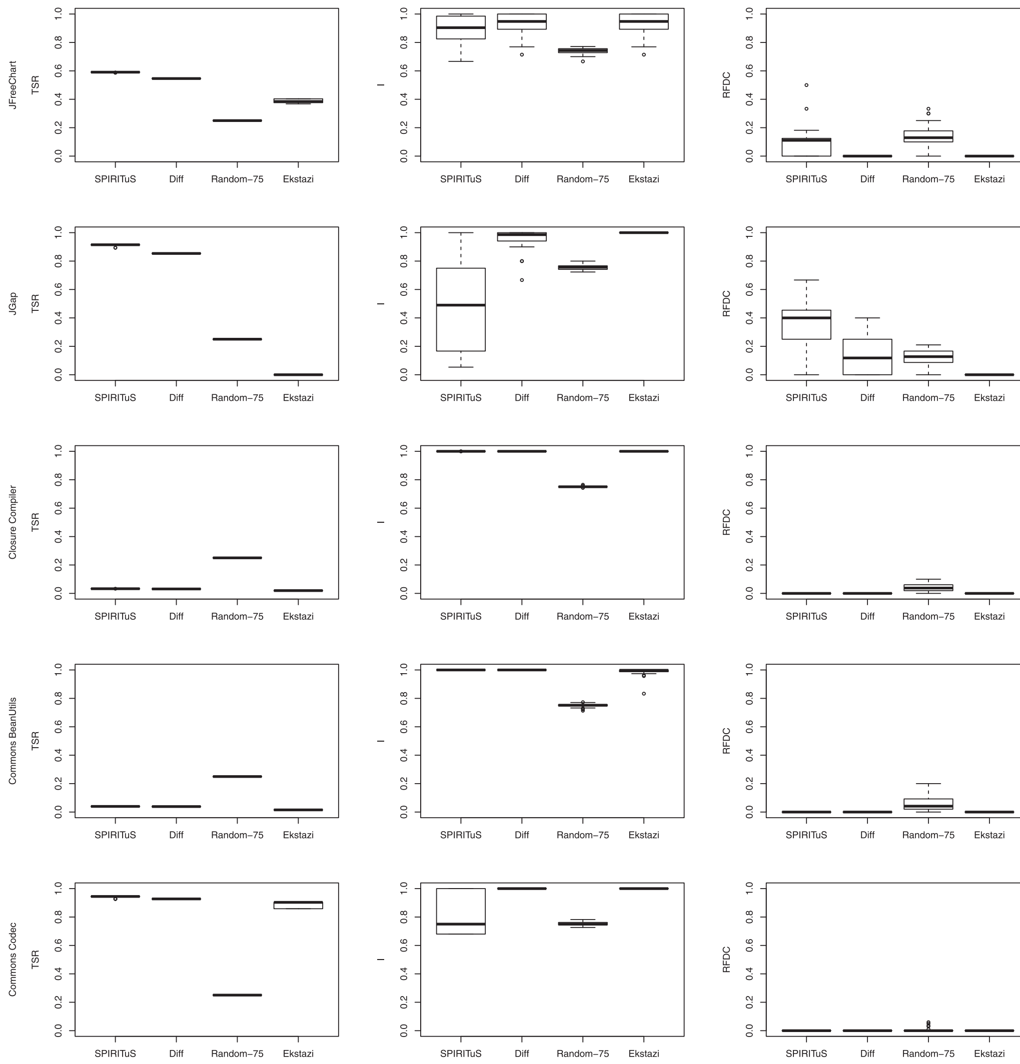
SPIRITuS (SimPle Information Retrieval regressIon Test Selection) is an information-retrieval based-approach for regression test case selection. We have presented the main characteristics of SPIRITuS and its empirical assessment on 14 open-source Java programs. We compared SPIRITuS with three competing approaches: Diff, Random-75, and Ekstazi. Our experimental results suggest that if a developer/tester has a quite limited testing budged he/she can use SPIRITuS because it selects a number of test cases significantly smaller than the number of test cases the other approaches/competitors select at the price of a slight reduction in fault detection capability. However, it is the tester who makes the final decision if such a slight reduction is tolerable.

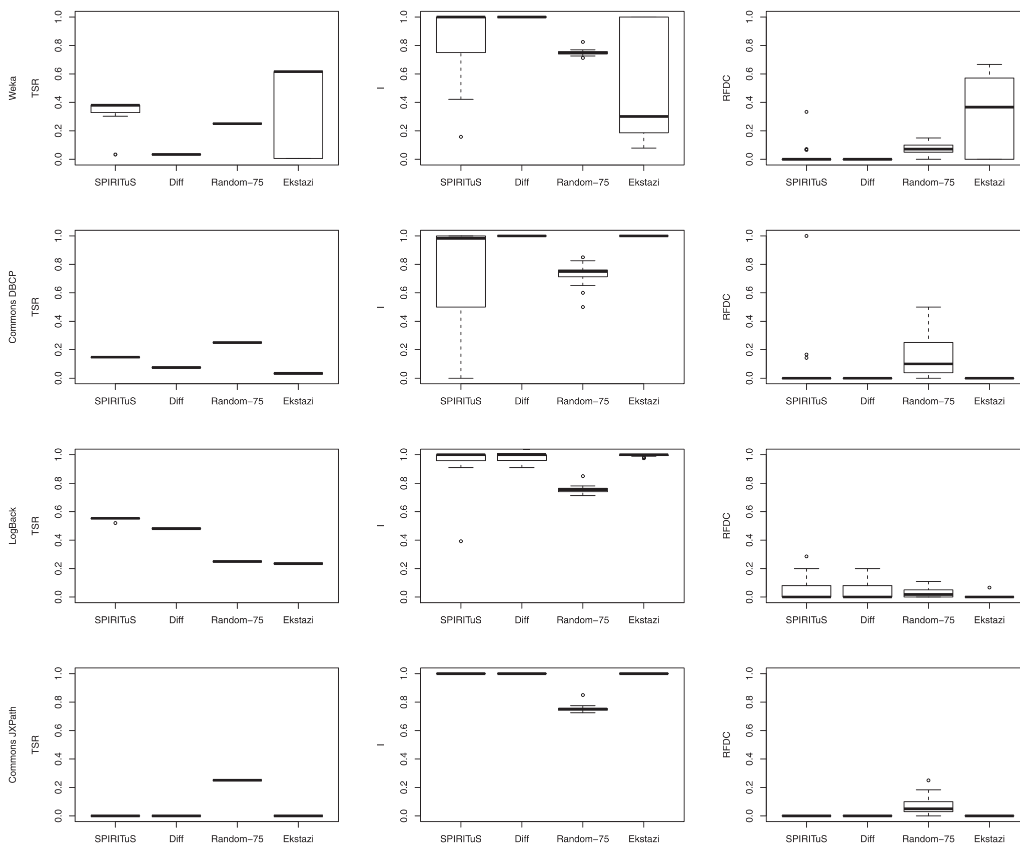
We also reported evidence that while SPIRITuS is better than the baseline approaches in 24 cases out of 42, while there is not a clear winner in all the other cases. Finally, we observed that SPIRITuS outperforms the other approaches when the average number of test cases covering a modified method increases and the number of commits between P and P' is low. In this scenario, SPIRITuS selects a lower number of test cases while preserving the fault detection effectiveness as the baseline approaches. Interestingly, this information can be easily derived before applying SPIRITuS.

Appendix A. Box plots for each experimental object

In this appendix, we report the boxplots for TSR, I and RFDC grouped by method and experimental object.


Commons Math
Commons Lang
Commons Configuration
Commons IO
Spring Context
TSR
I
RFDC
SPIRITuS
Diff
Random−75
Ekstazi
0.0
0.2
0.4
0.6
0.8
1.0


JFreeChart
JGap
Closure Compiler
Commons BeanUtils
Commons Codec
TSR
I
RFDC
SPIRITuS
Diff
Random−75
Ekstazi



References

[1] S. Yoo, M. Harman, Regression testing minimization, selection and prioritization: a survey, Softw. Test. Verif. Reliab. 22 (2) (2010) 67–120.
[2] S. Biswas, R. Mall, M. Satpathy, S. Sukumaran, Regression test selection techniques: A survey, Informatica 35 (3) (2011) 289–321.
[3] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, L. Zhang, B. Xie, Test case prioritization for compilers: A text-vector based approach, Proceedings of the International Conference on Software Testing, Verification and Validation, (2016), pp. 266–277.
[4] A. Marchetto, M.M. Islam, W. Asghar, A. Susi, G. Scanniello, A multi-objective technique to prioritize test cases, IEEE Trans. Softw. Eng. 42 (10) (2016) 918–940.
[5] A. Gotlieb, D. Marijan, FLOWER: optimal test suite reduction as a network maximum flow, Proceedings of the International Symposium on Software Testing and Analysis, (2014), pp. 171–180.
[6] L. De Souza, P. de Miranda, R. Prudencio, F. de Barros, A multi-objective particle swarm optimization for test case selection based on functional requirements coverage and execution effort, Proceedings of International Conference on Tools with Artificial Intelligence, IEEE, 2011, pp. 245–252.
[7] A. Panichella, R. Oliveto, M. Di Penta, A. De Lucia, Improving multi-objective test case selection by injecting diversity in genetic algorithms, IEEE Trans. Softw. Eng. 41 (4) (2015) 358–383.
[8] S. Mirarab, S. Akhlaghi, L. Tahvildari, Size-constrained regression test case selection using multicriteria optimization, IEEE Trans. Softw. Eng. 38 (4) (2012) 936–956.
[9] A. Srivastava, J. Thiagarajan, Effectively prioritizing tests in development environment, Proceedings of the International Symposium on Software Testing and Analysis, ACM, 2002, pp. 97–106.
[10] B. Busjaeger, T. Xie, Learning for test prioritization: An industrial case study, Proceedings of the International Symposium on Foundations of Software Engineering, ACM, New York, NY, USA, 2016, pp. 975–980.
[11] G. Rothermel, M.J. Harrold, J. von Ronne, C. Hong, Empirical studies of test-suite reduction, Softw. Test. Verif. Reliab. 12 (4) (2002) 219–249.
[12] G. Rothermel, M.J. Harrold, Analyzing regression test selection techniques, IEEE Trans. Softw. Eng. 22 (8) (1996) 529–551.
[13] S. Elbaum, A.G. Malishevsky, G. Rothermel, Test case prioritization: a family of empirical studies, IEEE Trans. Softw. Eng. 28 (2) (2002) 159–182.
[14] E. Engström, P. Runeson, M. Skoglund, A systematic review on regression test selection techniques, Inf. Softw. Technol. 52 (1) (2010) 14–30.
[15] F.I. Vokolos, P.G. Frankl, Pythia: a regression test selection tool based on textual differencing, Proceedings of the International Conference on Reliability Quality and Safety of Software Intensive Systems, (1997).
[16] M. Gligoric, L. Eloussi, D. Marinov, Practical regression test selection with dynamic file dependencies, Proceedings of the International Symposium on Software Testing and Analysis, ACM, 2015, pp. 211–222.
[17] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, D. Marinov, An extensive study of static regression test selection in modern software evolution, Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, 2016, pp. 583–594.
[18] G. Rothermel, M.J. Harrold, Empirical studies of a safe regression test selection technique, IEEE Trans. Softw. Eng. 24 (6) (1998) 401–419.
[19] J.H. Andrews, L.C. Briand, Y. Labiche, Is mutation an appropriate tool for testing experiments? Proceedings of the International Conference on Software Engineering, ACM, 2005, pp. 402–411.
[20] A. Shi, A. Gyori, M. Gligoric, A. Zaytsev, D. Marinov, Balancing trade-offs in test-suite reduction, Proceedings of the International Symposium on Foundations of Software Engineering, FSE 2014, ACM, 2014, pp. 246–256.
[21] H. Do, G. Rothermel, On the use of mutation faults in empirical assessments of test case prioritization techniques, IEEE Trans. Softw. Eng. 32 (9) (2006) 733–752.
[22] H. Do, S. Mirarab, L. Tahvildari, G. Rothermel, An empirical study of the effect of time constraints on the cost-benefits of regression testing, Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, 2008, pp. 71–82.
[23] D. Hao, L. Zhang, L. Zhang, G. Rothermel, H. Mei, A unified test case prioritization approach, ACM Trans. Softw. Eng. Methodol. 24 (2) (2014) 10:1–10:31.
[24] S. Romano, G. Scanniello, SMUG: a Selective MUtant Generator tool, Proceedings of the 39th International Conference on Software Engineering Companion, IEEE Press, 2017, pp. 19–22.
[25] C.D. Manning, P. Raghavan, H. Schütze, An Introduction to Information Retrieval, Cambridge University Press, England, 2009.
[26] R. Baeza-Yates, B. Ribeiro-Neto, Modern Information Retrieval, Addison-Wesley, 1999.
[27] L.J. White, H.K.N. Leung, A firewall concept for both control-flow and data-flow in regression integration testing, Proceedings of the Conference on Software Maintenance, (1992), pp. 262–271.
[28] Q.D. Soetens, S. Demeyer, A. Zaidman, Change-based test selection in the presence of developer tests, Proceedings of the Conference on Software Maintenance and Reengineering, (2013), pp. 101–110.
[29] Q.D. Soetens, S. Demeyer, Cheopsj: Change-based test optimization, Proceedings of the European Conference on Software Maintenance and Reengineering, IEEE Computer Society, 2012, pp. 535–538.
[30] K.F. Fischer, F. Raji, A. Chruscicki, A methodology for retesting modified software, Proceedings of the the National Telecommunications Conference, (1981), pp. 1–6.
[31] G. Rothermel, M.J. Harrold, A safe, efficient regression test selection technique, ACM Trans. Softw. Eng. Methodol. 6 (2) (1997) 173–210.
[32] M. Gligoric, R. Majumdar, R. Sharma, L. Eloussi, D. Marinov, Regression test selection for distributed software histories, Proceedings of the International Conference on Computer Aided Verification, (2014), pp. 293–309.
[33] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, Information retrieval models for

recovering traceability links between code and documentation, Proceedings of the IEEE International Conference on Software Maintenance, IEEE Computer Society, 2000, pp. 40–51.
[34] M.J. Harrold, J.A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S.A. Spoon, A. Gujarathi, Regression test selection for java software, Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications, ACM, 2001, pp. 312–326.
[35] T.L. Graves, M.J. Harrold, J. Kim, A. Porters, G. Rothermel, An empirical study of regression test selection techniques, Proceedings of the International Conference on Software Engineering, (1998), pp. 188–197.
[36] X. Ren, F. Shah, F. Tip, B.G. Ryder, O. Chesley, Chianti: a tool for change impact analysis of java programs, Proceedings of the Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, ACM, 2004, pp. 432–448.
[37] L. Zhang, M. Kim, S. Khurshid, Localizing failure-inducing program edits based on spectrum information, Proceedings of the International Conference on Software Maintenance, IEEE Computer Society, 2011, pp. 23–32.
[38] R.K. Saha, L. Zhang, S. Khurshid, D.E. Perry, An information retrieval approach for regression test prioritization based on program changes, Proceedings of the International Conference on Software Engineering, IEEE Press, Piscataway, NJ, USA, 2015, pp. 268–279.
[39] A. Corazza, V. Maggio, G. Scanniello, Coherence of comments and method implementations: a dataset and an empirical investigation, Softw. Q. J. (2016) 1–27. Cited By 0; Article in Press
[40] B. Fluri, M. Wursch, H.C. Gall, Do code and comments co-evolve? on the relation between source code and comment changes, Proceedings of the 14th Working Conference on Reverse Engineering, WCRE '07, IEEE Computer Society, 2007, pp. 70–79.
[41] Z.M. Jiang, A.E. Hassan, Examining the evolution of code comments in postgresql, Proceedings of the 2006 International Workshop on Mining Software Repositories, MSR '06, ACM, 2006, pp. 179–180.
[42] J. Gosling, B. Joy, G.L. Steele, G. Bracha, A. Buckley, The Java Language Specification, Java SE 8 Edition, 1st, Addison-Wesley Professional, 2014.
[43] B. Dit, M. Revelle, M. Gethers, D. Poshyvanyk, Feature location in source code: a taxonomy and survey, J. Softw. Evol. Process 25 (1) (2013) 53–95.
[44] G. Scanniello, A. Marcus, D. Pascale, Link analysis algorithms for static concept location: an empirical assessment, Empirical Softw. Engg. 20 (6) (2015) 1666–1720.
[45] A. Abadi, M. Nisenson, Y. Simionovici, A traceability technique for specifications, Proceedings of the International Conference on Program Comprehension, ICPC '08, IEEE CS Press, Washington, DC, USA, 2008, pp. 103–112.
[46] S. Wang, D. Lo, Z. Xing, L. Jiang, Concern localization using information retrieval: An empirical study on linux kernel, Proceedings of Working Conference on Reverse Engineering, WCRE, IEEE Computer Society, 2011, pp. 92–96.
[47] A. Marcus, J.I. Maletic, Recovering documentation-to-source-code traceability links using latent semantic indexing, Proceedings of the ICSE, (2003), pp. 125–137.
[48] S.C. Deerwester, S.T. Dumais, T.K. Landauer, G.W. Furnas, R.A. Harshman, Indexing by latent semantic analysis, J. Am. Soc. Inf. Sci. 41 (6) (1990) 391–407.
[49] S.K. Lukins, N.A. Kraft, L.H. Etzkorn, Bug localization using latent Dirichlet allocation, Inf. Softw. Technol. 52 (9) (2010) 972–990.
[50] D. Dig, C. Comertoglu, D. Marinov, R. Johnson, Automated detection of refactorings in evolving components, Proceedings of European Conference on Object-Oriented Programming, Springer-Verlag, 2006, pp. 404–428.
[51] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering, Springer, 2012.
[52] O.J. Dunn, Multiple comparisons among means, J. Am. Stat. Assoc. 56 (1961) 52–64.
[53] P. Ellis, The Essential Guide to Effect Sizes: Statistical Power, Meta-Analysis, and the Interpretation of Research Results, Cambridge University Press, 2010.
[54] N. Cliff, Dominance statistics: Ordinal analyses to answer ordinal questions, Psychol. Bull. 114 (3) (1993) 494–509.
[55] J. Romano, J.D. Kromrey, J. Coraggio, J. Skowronek, Appropriate statistics for ordinal level data : should we really be using t-test and Cohen's d for evaluating group differences on the nsse and other surveys? Proceedings of the Annual meeting of the Florida Association of Institutional Research, (2006).
[56] M. Gligoric, L. Eloussi, D. Marinov, Ekstazi: Lightweight test selection, Proceedings of the International Conference on Software Engineering, Volume 2 IEEE Press, 2015, pp. 713–716.
[57] V. Arnaoudova, L.M. Eshkevari, M. Di Penta, R. Oliveto, G. Antoniol, Y. Guéhéneuc, REPENT: analyzing the nature of identifier renamings, IEEE Trans. Softw. Eng. 40 (5) (2014) 502–532.
[58] Z. Xing, E. Stroulia, Refactoring detection based on UMLDiff change-facts queries, Proceedings of Working Conference on Reverse Engineering, WCRE, IEEE Computer Society, 2006, pp. 263–274.
[59] S.L. Pfleeger, W. Menezes, Marketing technology to software practitioners, IEEE Softw. 17 (1) (2000) 27–33.