Scope-aided Test Prioritization, Selection and Minimization for Software Reuse

Breno Miranda[a,b,*], Antonia Bertolino[b]

[a]Dipartimento di Informatica
Università di Pisa
Largo B. Pontecorvo, 3
56127 Pisa, Italy
[b]ISTI - CNR
Via Moruzzi 1
56124 Pisa, Italy

Abstract

Software reuse can improve productivity, but does not exempt developers from the need to test the reused code into the new context. For this purpose, we propose here specific approaches to white-box test prioritization, selection and minimization that take into account the reuse context when reordering or selecting test cases, by leveraging possible constraints delimiting the new input domain scope. Our scope-aided testing approach aims at detecting those faults that under such constraints would be more likely triggered in the new reuse context, and is proposed as a boost to existing approaches. Our empirical evaluation shows that in test suite prioritization we can improve the average rate of faults detected when considering faults that are in scope, while remaining competitive considering all faults; in test case selection and minimization we can considerably reduce the test suite size, with small to no extra impact on fault detection effectiveness considering both in-scope and all faults. Indeed, in minimization, we improve the in-scope fault detection effectiveness in all cases.

Keywords: In-scope entity, test case selection, test case prioritization, test suite minimization, test of reused code, testing scope

1. Introduction

Since the early days of the software engineering discipline, reuse has been advocated as a key solution to face the "software crisis" and improve software quality while reducing time-to-market [31, 39, 40]. Moreover, integrating pre-existing components or portions of code that had already been widely tested and used seemed a prelude to easily achieve more reliable systems, as showed e.g. in [33].

*Corresponding author. *E-mail address:* breno.miranda@isti.cnr.it

Postprint published as DOI 10.1016/j.jss.2016.06.058

However decades of research and practice in software reuse have not yet fully realized such optimistic expectations. Along with several success stories, the literature reports about many failures [34, 27], some also with catastrophic consequences as the often cited disasters of Therac 25 accidents [28] and of Ariane 5 Flight 501 [30], both eventually attributed to bad test reuse practices.

As Weyuker warned in [46], *developers reusing a component need to do considerable testing to ensure the software behaves properly in its new environment.* In this regard, the results from a recent survey among developers from different companies on the current practice of testing of reused software [45] are dismaying: to the question whether they tested a component in any way before reusing it, only 43% of the interviewed developers answered yes, while 41% did it rarely and 16% not at all. From the interviews it emerged that a reason at the origin of such risky behavior was the lack of proper and simple-to-use tools, in particular for facilitating the testing of code that has been written by someone else. The authors report that many developers would like to see a tool that could give them some measure on how well they have tested a given function, class or code portion, as for example code coverage tools.

Code coverage measures are widely used in testing practice as an adequacy criterion [16]: given a set of program entities to be covered (they could be statements, branches, paths, functions, and so on), a program is not considered to be adequately tested until all such entities (or a given percentage thereof) have been executed at least once. Moreover, test coverage tools provide guidance as to what parts of a program should be exercised to augment a not adequate test suite. Indeed, although the relationship between code coverage and test suite effectiveness is debated, see e.g. [20], there is consensus that having coverage information is important as evidently a test suite can hardly find bugs in code that is never executed.

When the source code is available, as in the case of reuse from internal repository [41], measuring the test coverage of reused code would be useful as well for developers [45]. The problem is that when testing reused software, traditional ways to measure test coverage might produce meaningless information: in fact, not all entities might be of interest when a code is reused in a new context and environment. We refer to the latter as a testing *scope*.

In a recent paper [32], we proposed to adapt test coverage measures to the specific reuse scope. Reaching 100% test coverage might not be a sensible target from within a certain scope in which some parts of a reused code would never be invoked. A more useful information for testers would be to know how much coverage they could achieve within their reuse scope. Thus, instead of measuring coverage as the ratio between the entities covered and the total number of entities, in [32] we suggested to measure coverage of reused code as the ratio between the entities covered and those that could potentially be exercised in the new scope. We introduced the term of *in-scope entities* to refer to the entities in the reused code that are exercised from the new context. The remaining ones are referred to as the *out-of-scope entities.* Out-of-scope entities are not to be confused with infeasible entities [13]: they are perfectly feasible, i.e. there exist test inputs that exercise them, but a user would never or very unlikely invoke such inputs in the new reuse context.

A large portion of testing research aims at reducing testing effort and time. Many different techniques have been developed for test prioritization, selection and minimization: all such techniques take in input an existing pool of test cases and produce a test suite that respectively reorders, reduces or tries to minimize the original set, so to reduce testing costs. More cost-effective testing approaches remain an exigency also for reused software. In the cited survey on reuse and testing [45], 60% of the developers claimed that verification and validation was the first thing that was reduced in case of time shortage during a project.

In this paper *we propose to use scope information to improve test prioritization, selection and minimization for the purpose of testing reused code.* We call our approach *scope-aided* testing, as we do not propose novel techniques, but take into account the reuse scope of entities when applying existing approaches for test prioritization, selection and minimization. Although based on a simple, intuitive idea, our approach may provide developers with guidance in the testing of reused code.

In the testing literature, test prioritization, selection and minimization have been mostly investigated for regression testing [47], i.e. for retesting a software code after modifications. Here we address test prioritization, selection and minimization for reuse testing, i.e. for retesting a software component or a piece of reused code (possibly unchanged) from within a new context in which it is invoked under a different usage profile.

Ravichandran and Rothenberger [41] distinguished among three reuse strategies: White-Box Reuse with internal components, Black-Box Reuse with internal components, and Black-Box Reuse with component markets. More precisely, our approach can be applied to both the first and the second strategy, in which the source code is available. Then, in Black-Box Reuse with internal components the modification of reused code is not allowed, and thus we apply the approaches presented in the following without considering any modifications. In the case of White-Box Reuse, code could be modified, in such case in testing we should consider both the change of context and of code. We can apply scope-aided testing taking into account also the changes, i.e., we can use any regression testing strategy empowered with scope information. For clarity of exposition, since the focus of this work is on the new concept of scope-driven approaches, the examples and studies in the paper are not modification-aware.

To the best of our knowledge, there exist no previous work specifically addressing test prioritization, selection and minimization for reused software. We define scope-aided test prioritization, selection and minimization, and conduct an empirical evaluation of them by comparing several metrics in the two cases of using scope to boost existing techniques vs. applying the original techniques unchanged. Precisely, we applied "scope-aid" to boost total and additional greedy, similarity-based and search-based prioritization; greedy additional selection; and the GE algorithm for minimization proposed in [6]. The results vary across the subject programs, and we provide a detailed report in the following. However overall they are encouraging in that scope may help detect relevant faults faster, and can be used to derive a reduced test suite in reuse contexts.

The paper is structured as follows: we introduce a motivating example in Section 2 and some general definitions in Section 3. We then introduce more detailed definitions for scope-aided prioritization, selection and minimization in Section 4: for each, we provide first a definition of scope-aided approach and then illustrate the approach on the example anticipated in Section 2. We then describe the settings of our empirical evaluations in Section 5, including the Research Questions, the study subjects, tasks, procedures and metrics. Results are reported and illustrated separately for prioritization, selection and minimization in Sections 6, 7, and 8, respectively. We discuss costs and benefits of the approach in Section 9, and threats to the validity of our study in Section 10. Related Work (Section 11) and Conclusions (Section 12) complete the paper.

2. Motivating Example

To illustrate our scope-aided testing approach, let us consider the small program displayed in the leftmost column of Table 1, which contains two functions, `Example1` and `Example2`. Assuming a statement-coverage approach, Table 1 also provides information regarding a set of test cases T = [TC$_1$,TC$_2$,TC$_3$,TC$_4$,TC$_5$] used to exercise the example code. Each test case is displayed along with the values used for the input variables (`x`, `y`, and `z`) as well as the statements covered (please ignore for the moment the last two lines, whose meaning is explained later on).

As highlighted in Table 1, we assume two faulty statements in the function `Example1`. The first one can only be triggered if Statement 2 is executed: hence only test cases with `x` bigger than 0 are likely to reveal this fault. The second fault, on the other hand, can only be triggered when Statement 5 is traversed: hence only test cases with `x` equal to or less than 0 are likely to expose this fault.

White-box approaches for test case prioritization, selection, and minimization evaluate test cases based on how the entities in the program flowgraph are covered. For example, the adoption of the *additional* prioritization strategy would yield a prioritized test suite TS_p = [TC$_2$,TC$_1$,TC$_4$,TC$_5$,TC$_3$]. A test case selection heuristic based on the *additional* greedy algorithm would produce the test suite TS_s = [TC$_2$,TC$_1$,TC$_4$,TC$_5$]; And the test minimization approach as proposed by [6] would provide a minimized test suite TS_m = [TC$_2$,TC$_1$,TC$_4$,TC$_5$].

Let us now assume that the aforementioned code is going to be reused in a scope in which we know that `x` will always be bigger than 0 and the function `Example2` is no longer used. In Table 1, the statements exercised in such context are highlighted in grey: we see that Statement 5 would never be reached and so *fault 2* would never be triggered. In our terminology, *fault 2* is said to be out-of-scope, whereas *fault 1* that can be triggered under the known constraint is an in-scope fault.

Table 1: Statement coverage achieved by the test cases used to exercise the sample code

Code	**TC1** (x=1,y=1)	**TC2** (x=-1,y=-1)	**TC3** (x=1,y=-1)	**TC4** (z=0)	**TC5** (z=10)
Example1(x, y):					
1. if x >0:	✓	✓	✓		
2. s2 //fault 1	✓		✓		
else:					
3. s3		✓			
4. s4		✓			
5. s5 //fault 2		✓			
6. if y >0:	✓	✓	✓		
7. s7	✓				
else:					
8. s8		✓	✓		
9. s9		✓	✓		
Example2(z):					
10.if z == 0:				✓	✓
11. s11				✓	
else:					
12. s12					✓
#st. covered	4	7	5	2	2
#in-scope st.	4	4	5	0	0
#out-of-scope st.	0	3	0	2	2

Our scope-aided testing approach aims at exploiting this kind of context-related information to bias test case prioritization, selection, and minimization so to give priority to those test cases that are more likely to reveal in-scope faults.

Scope-aided testing is not an approach per se, but a boost to existing prioritization, selection, and minimization approaches. For example, the application of scope-aided prioritization, when used to boost the *additional* approach, would provide a prioritized test suite $TS_{p'}$ = [TC$_3$,TC$_1$,TC$_2$,TC$_4$,TC$_5$]. Any of the prioritized test suites (either scope-aided or not scope-aided) would trigger the two faults, even though *fault 2* is very unlikely to manifest itself within the context scope in which `x` is always bigger than 0. However, the adoption of a scope-aided prioritized test suite brings the benefit of revealing *fault 1*, the critical one, faster.

The adoption of scope-aided selection to improve the selection heuristic would provide the test suite $TS_{s'}$ = [TC$_3$,TC$_1$], whereas its adoption to enhance the minimization approach proposed by [6] would yield the test suite $TS_{m'}$ = [TC$_1$,TC$_2$]. In both cases the scope-aided approach would preserve the fault detection capability of the original test suites (when considering in-scope faults) while bringing the benefit of yielding smaller test suites.

3. General Definitions

This section introduces the basic concepts and definitions related to our approach. We start by defining the notions of *scope* and *in-scope entity*, already anticipated in the introduction. When specifically referring to a testing context and environment, we also call it a testing scope.

Definition 1 (*(Testing) Scope*): A subset of the (testing) input domain. More formally, given the input domain $\mathcal{D}$ of a program P, and given a set C of constraints over $\mathcal{D}$, a (testing) scope $\mathcal{S}$ is defined by the set of (test) input values to program P that satisfy the constraints C.

In the above definition, the constraints can be as formal as algebraic expressions over P input variables, or general properties delimiting the input domain $\mathcal{D}$. Notice that defining the testing scope is *not* part of our approach. Rather, it presupposes that the information regarding the specific testing context is available.

Definition 2 (*In-scope entities*): The set of entities relevant to a given scope. More formally, given a program P with entities $\{e_1, e_2, ..., e_n\}$ and a scope $\mathcal{S}$, the set of in-scope entities with regards to S is $\mathcal{E}_s = \{e_{i_1}, e_{i_2}, ..., e_{i_n}\}$ such that $\forall e_{i_j}$ there exists some input $v \in S$ that covers them.

Definition 3 (*Out-of-scope entities*): The set of entities that are not relevant to a given scope (they are not covered by any input $v \in S$).

For the purpose of exposition in the following we will refer to relevant, or in-scope, faults. This notion would refer to those faults that can be triggered by some inputs within the scope. In real life this notion cannot be easily defined since doing this would require to exhaustively exercise the program onto the whole scope $\mathcal{S}$, which is not feasible except for trivial programs. Therefore the notion of relevant faults can only be defined in probabilistic terms.

Definition 4 (*In-scope fault*): A fault that is likely to manifest itself as a failure under the scope inputs subset.

Definition 5 (*Out-of-scope fault*): A fault that is not likely to manifest itself as a failure under the scope inputs subset.

To make the above definitions meaningful, we need to provide a definition for "likely". In the context of this work we evaluated the likelihood of a given fault to manifest itself as a failure by using randomly generated test suites and mutation testing (as detailed next in Section 5.2).

4. Scope-aided Testing Approaches

White-box approaches for test case prioritization, selection, and minimization evaluate test cases based on how the entities in the program flowgraph are covered. Our scope-aided approach enhances existing traditional techniques by taking into account the relevance of the entities to be covered with respect to possible constraints delimiting the input domain scope. It is proposed as a boost to existing approaches for focussing on in-scope faults when a code is reused in a different context.

The core of our approach lies in the identification of in-scope entities. This is achieved by the following steps:

> **1) Constraints identification:** As the first step of our approach, we use the information regarding the specific testing scope to identify the input domain constraints that will be further used to identify the in-scope entities. As said, the input domain constraints provided could vary, they could be coarse grained such as a list of functions that are expected to be used in that specific scope; or fine grained such as the precise range of values expected to be used by a given variable. The more information is provided, the more precise is the identification of the in-scope entities. In the example from the motivating scenario in Section 2, the constraints provided were the two facts that, in the new test context, (i) variable `x` would always be bigger than 0; and (ii) function `Example2` would be no longer used. Other constraints such as the range of values expected for the variable `y`, for example, could also be provided.
>
> **2) In-scope entities identification:** Different approaches could be adopted to identify the entities that are relevant under the input domain constraints collected in the first step of our approach. For example, one could apply a reachability algorithm on the static call graph of the given program. Even though this is an undecidable problem [35], there exist algorithms capable of generating approximated solutions. We decided to use Dynamic Symbolic Execution (DSE) since it has shown to be a powerful approach to analyze the code dynamically and guide its exploration based on the input domain constraints [38, 7]. Our decision was influenced by the fact that DSE is very actively investigated and several tools are available. For this work, we adopted KLEE [3], a well-known symbolic execution tool capable of automatically generating tests that achieve high coverage even for complex and environmentally-intensive programs. To guide the DSE exploration, we provide the input domain constraints (collected in the previous step of our approach) to KLEE. When it is run on the target code, KLEE tries to explore all paths, finding concrete test inputs to exercise them. The set of in-scope entities consists of those exercised by the

DSE-generated test cases[1]. Indeed, if some entities are not exercised by the DSE-generated test cases it is because they are not reachable under the input domain constraints provided. As said, in Table 1 the in-scope statements are those highlighted in grey.

Having followed these steps, the set of in-scope entities is used in input to our approach. In the next three subsections we detail how they are used for scope-aided prioritization, selection, and minimization, respectively. In each subsection, we first define the problem we are addressing and then illustrate an instantiation of our scope-aided approach to support the problem at hand (using the motivating example depicted in Section 2). For each example, we first review how a considered traditional technique works, and then explain how the scope-aided approach could be adopted to improve testing of reused software.

4.1. Scope-aided Prioritization

Test case prioritization consists into reordering the test cases saved into a test suite so that potential faults can be detected as early as possible. Prioritization per se does not involve a reduction of the test suite size, but only a modification of its execution ordering. This does not directly imply a cost reduction, however brings two important advantages. If fault detection is anticipated, debugging can start earlier. Moreover, should testing activity be terminated prematurely (e.g. because of budget restrictions), then having executed the test cases in the prioritized order ideally ensures that the available time has been well spent to perform the most effective ones.

4.1.1. Definition of scope-aided prioritization

More formally, the test case prioritization problem is commonly defined [42] as follows:

Definition 6. (*Prioritization Problem*):

Given: *A test suite T; the set PT of permutations of T; a function f from PT to the real numbers $\mathcal{R}$*

Problem: *Find $T' \in PT$ such that $\forall T''$: ($T'' \in PT$) and ($T'' \neq T'$): [$f(T') \geq f(T'')$]*

In the above definition f is a given function that assigns some award value to a test suite order. Ideally, such function would refer to fault detection rate: the earlier the faults are detected, the more preferable a test suite order is. Unfortunately in reality we cannot know in advance which test case detects which fault, so prioritization approaches can only be based on surrogate criteria [47] that are demonstrated to give good approximations for fault detection effectiveness.

[1]Note that we "replay" the test cases generated using the original program and use gcov so to get accurate coverage achieved by the DSE-generated test cases.

Both black-box and white-box surrogate criteria have been devised. In this work, we focus on white-box prioritization approaches in which the ordering is done according to how the entities in the program flow graph are covered. Several different methods have been introduced providing different instantiations for the "how" in the sentence above: in additional greedy ordering [43] a test case t_i will precede another test case t_j if t_i covers more yet uncovered entities than t_j; in similarity-based prioritization [23] instead, t_i will go before t_j if t_i covers a flow graph path that is more dissimilar from all covered paths so far than t_j covered path; in search-based prioritization [29] t_i is preferred over t_j if t_i provides a higher increase for the adopted fitness function than t_j does.

We now introduce the notion of Scope-aided Prioritization in analogous way to that of Prioritization as follows:

Definition 7. (***Scope-aided Prioritization Problem***):

Given*: A test suite T; the set PT of permutations of T; a testing scope $\mathcal{S}$; a function f_s from PT and $\mathcal{S}$ to the real numbers $\mathcal{R}$*

Problem*: Find $T' \in PT$ such that $\forall T''$: $(T'' \in PT)$ and $(T'' \neq T')$: $[f_s(T') \geq f_s(T'')]$*

Thus the problem of scope-aided prioritization is the same as prioritization, but we use an objective function f_s that awards those orderings that would reveal in-scope faults faster (more precisely, we use coverage of in-scope entities as a surrogate criterion for detection of in-scope faults).

The scope-aided prioritization approach can be adopted to boost a wide spectrum of traditional test case prioritization techniques. For the experiments reported in the following, we considered two coverage-based approaches (*total* and *additional*), one similarity-based approach, and one search-based approach.

4.1.2. The scope-aided approach applied to a prioritization example

In this section we illustrate our approach being applied to boost a coverage-based prioritization technique. Both *additional* and *total* strategies have been proposed [17]. For illustrative purposes, next we instantiate our approach on the *additional* strategy (application to total would be quite similar).

Algorithm 1 depicts the additional strategy applied for test case prioritization. On each iteration, the test case that yields the highest coverage is selected (line 3) and the coverage information of the remaining test cases is updated (line 6) to reflect their coverage with respect to the not yet covered entities.

When multiple test cases cover the same number of not yet covered entities, an additional rule is needed to decide which one of these test cases will be selected. One possibility would be to simply pick one of the tied test cases randomly, but this would imply having a non-deterministic output. Our implementation of *getNextTestCase* sorts the tied test cases in ascending order according to their IDs in order to guarantee a deterministic output.

After prioritizing a subset of the test cases it is possible to reach a point in which all the entities are covered and the remaining test cases cannot con-

Algorithm 1: Prioritization (additional strategy)

```
Input: $T$                                   /*the test suite to be prioritized*/
       $coverageInfo$   /*list of entities covered by each test case from T*/
Output: $T'$                                         /*a permutation of T*/
1 $T' \longleftarrow [\,]$                  /*$T'$ is initialized as an empty list*/
2 while $thereAreTestCasesToBePrioritized(T, T', coverageInfo)$ do
3     $selectedTestCase \longleftarrow getNextTestCase(T, coverageInfo)$   /*selects the
        test case that yields the greatest coverage*/
4     $add(selectedTestCase, T')$
5     $remove(selectedTestCase, T)$
6     $updateCoverageInformation(coverageInfo, selectedTestCase)$   /*adjusts the
        coverage information of the remaining test cases to indicate their
        coverage of entities not yet covered*/
  end
```

tribute to increase coverage anymore. At this point, different approaches could be adopted to order the remaining test cases. One possibility could be resetting the coverage vectors of the remaining test cases and reapplying the additional strategy algorithm as done in [42]; another possibility could be adopting a different approach (e.g., the total strategy) to prioritize the remaining tests. Algorithm 1 does not adopt alternative strategies and keeps selecting the test cases in the same way, which basically means that when the maximum coverage is achieved, the remaining test cases are ordered according to their IDs.

When applied to the motivating example in Table 1, the first test case that Algorithm 1 selects — the one that achieves the highest coverage — is TC$_2$ that covers seven (1, 3, 4, 5, 6, 8, and 9) out of 12 possible statements. Then, it looks for the next test case that achieves the highest coverage with respect to the yet to be covered statements (2, 7, 10, 11, and 12). For the next choice, three test cases are tied (TC$_1$, TC$_4$, and TC$_5$) covering two statements each and TC$_1$ is the one selected (as explained before). The list of uncovered statements is then updated (10, 11, and 12). The next test case chosen is TC$_4$ and then TC$_5$. At this point, 100% of the statements have been covered already and the remaining test cases are ordered according to their IDs. The prioritized test suite returned by Algorithm 1 would be TS_p = [TC$_2$,TC$_1$,TC$_4$,TC$_5$,TC$_3$].

Scope-aided prioritization adopting the additional strategy is depicted in Algorithm 2. It is analogous to Algorithm 1 with the following modifications:

1. The set of in-scope entities, received as input, is used by *getNextTestCase* to decide which test case achieves the highest coverage;
2. If multiple test cases cover the same number of not yet covered in-scope entities, *getNextTestCase* tries to solve the tie by returning the test case that achieves the highest additional coverage when considering all the entities. If the tie persists, then the tied test cases are sorted in ascending order according to their IDs.

Algorithm 2: Scope-aided Prioritization (additional strategy)

```
   Input: $T$                                  /*the test suite to be prioritized*/
          $coverageInfo$   /*list of entities covered by each test case from T*/
          $inscopeEntities$   /*list of entities relevant to the current scope*/
   Output: $T'$                                         /*a permutation of T*/
1  $T' \longleftarrow [\,]$                       /*$T'$ is initialized as an empty list*/
2  while $thereAreTestCasesToBePrioritized(T, T', coverageInfo, inscopeEntities)$
    do
3      $selectedTestCase \longleftarrow getNextTestCase(T, coverageInfo, inscopeEntities)$
        /*selects the test case that yields the greatest coverage of in-scope
        entities*/
4      $add(selectedTestCase, T')$
5      $remove(selectedTestCase, T)$
6      $updateCoverageInformation(coverageInfo, inscopeEntities, selectedTestCase)$
        /*updates the set of in-scope entities and adjusts the coverage
        information of the remaining test cases to indicate their coverage of
        entities not yet covered*/
   end
```

Modification 2 brings the advantage of making Algorithm 2 revert to the traditional prioritization approach (Algorithm 1) after all the in-scope entities are covered.

The second last line of Table 1 counts the number of statements covered by each test case considering only the ones that belong to the set of in-scope statements. When Algorithm 2 is applied, the first test case selected is TC_3 as it is the one that covers the highest number of in-scope statements. After TC_3 is selected, there is only one in-scope statement that needs to be covered (statement 7). TC_1 is then selected as it is the only test case that covers that statement. After TC_1 is selected, 100% coverage is achieved for the in-scope statements and the remaining test cases are ordered as explained before. The test suite prioritized by Algorithm 2 is then provided as: $TS_{p'}$ = [TC_3,TC_1,TC_2,TC_4,TC_5].

4.2. Scope-aided Selection

Test case selection deals with the problem of selecting a subset of test cases that will be used to test the software with respect to a given testing objective. The majority of the selection techniques are *modification-aware*, i.e., they seek to identify test cases that are relevant to some set of recent changes in the software under test (in such context, it may also be referred to as Regression Test case Selection).

However, test case selection can also be oriented towards different testing objectives: it can focus, for example, on selecting tests to exercise the parts of the software that are too expensive to fix after launch (risk-based test case selection); or it can focus on ensuring that the software is capable of completing the core operations it was designed to do (design-based test case selection) [26].

4.2.1. Definition of scope-aided selection

More formally, the problem of test case selection can be defined as follows:

Definition 8. (*Selection Problem*):

Given: *A program P; and a test suite T*

Problem: *Find a subset of T, T', with which to test P*

The problem of scope-aided selection is analogous to the selection problem and it aims at selecting a set of test cases to test a given program with respect to a testing scope. It is defined in analogous way to that of selection as follows:

Definition 9. (*Scope-aided Selection Problem*):

Given: *A program P; a test suite T; and a testing scope $\mathcal{S}$*

Problem: *Find a subset of T, T', such that $T' \in \mathcal{S}$, with which to test P*

In this work, because we focus on test case selection for reused software, which does not necessarily imply regression test selection, we use, for our instantiation example and for our exploratory study, a test selection technique that is not modification-aware. Yet our approach can also be applied to the regression test case selection problem.

4.2.2. The scope-aided approach applied to a selection example

To instantiate the scope-aided approach being applied to support a test case selection example, we refer to the following selection heuristic: use the greedy additional algorithm to repeatedly select the test case that covers the maximum number of uncovered entities until all entities are covered. This heuristic is depicted in Algorithm 3.

Algorithm 3: Selection (greedy additional heuristic)

```
Input: $T$              /*the test suite from which test cases can be selected*/
       $entities$                          /*list of entities to be covered*/
       $coverageInfo$        /*list of entities covered by each test from T*/
Output: $T'$       /*a subset of T, with which to test the target program*/
1 $T' \longleftarrow [\,]$                      /*$T'$ is initialized as an empty list*/
2 while $thereAreUncoveredEntities(T, entities, coverageInfo)$ do
3     $selectedTestCase \longleftarrow getNextTestCase(T, entities, coverageInfo)$
        /*selects the test case that covers the highest number of uncovered
        entities*/
4     $add(selectedTestCase, T')$
5     $updateUncoveredEntities(entities, selectedTestCase)$ /*removes the entities
        covered by the selected test case from the list of uncovered
        entities*/
  end
```

In Algorithm 3, the function *getNextTestCase* (line 3) behaves in the same way as that of Algorithm 1, that is, when multiple test cases cover the same number of not yet covered entities, it sorts the tied test cases in ascending order according to their IDs in order to guarantee a deterministic output.

Without the support of our approach, the first test case selected by Algorithm 3 is TC_2 as it covers the highest number of uncovered statements (7 out of 12). For the next choice, three test cases are tied (TC_1, TC_4, and TC_5) covering two statements each, and TC_1 is selected. After TC_1 is selected, three statements still need to be covered (10, 11, and 12) and TC_4 and TC_5 are tied again. After the tie is solved, TC_4 is chosen and then TC_5. When TC_5 is selected, 100% of the statements have been covered and test case selection stops producing the final test suite TS_s = [TC_2,TC_1,TC_4,TC_5].

The scope-aided selection makes use of the same algorithm. The only difference is that the list of entities to be covered, which is provided as input for Algorithm 3, contains the set of in-scope entities identified by our approach, rather than the full set of entities available in the target program. When supported by the scope-aided approach, TC_3 is the first test case chosen by Algorithm 3 as it covers the highest number of in-scope statements (we recall that the second last line of Table 1 counts the number of in-scope statements covered by each test case). After TC_3 is selected, statement 7 is the only statement yet to be covered. TC_1 is the next test case chosen as it covers statement 7. After TC_1 is selected, 100% coverage is achieved for the in-scope statements and the test case selection stops. The test suite produced by Algorithm 3 is then provided $TS_{s'}$ = [TC_3,TC_1].

4.3. Scope-aided Minimization

Test suite minimization is a technique that seeks to reduce as much as possible the test suite size by identifying and eliminating redundant test cases from it.

4.3.1. Definition of scope-aided minimization

More formally, test suite minimization [19] can be defined as follows:

Definition 10. (***Minimization Problem***):

Given*: A program P; a test suite T; a set of entities $\mathcal{E} = \{e_1, ..., e_n\}$ that must be exercised to provide the desired test coverage of P; and subsets of T: $\{T_1, ..., T_n\}$, each one associated with one of the e_i such that any of the test cases $t_j \in T_i$ can be used to test e_i*

Problem*: Find a representative set T' of test cases from T that satisfies all $e_i \in \mathcal{E}$*

The problem of scope-aided minimization is analogous to the minimization problem, but we aim at covering only the entities that belong to a given testing scope. The definition of Scope-aided Minimization, comparable to that of Minimization, is provided as follows:

Definition 11. (*Scope-aided Minimization Problem*):

***Given**: A program P; a test suite T; a set of entities $\mathcal{E} = \{e_1, ..., e_n\}$ that must be exercised to provide the coverage of P; a testing scope $\mathcal{S}$; a subset of in-scope entities $\mathcal{E}_s \in \mathcal{E}$; and subsets of T: $\{T_{s_1}, ..., T_{s_m}\}$, each one associated with one of the $e_i \in \mathcal{E}_s$ such that any of the test cases t_j belonging to T_{s_i} can be used to test e_i*

***Problem**: Find a representative set T' of test cases from T, such that $T' \in \mathcal{S}$, that satisfies all $e_i \in \mathcal{E}_s$*

The test suite minimization problem is equivalent to the well-known set covering problem which is NP-complete [24]. For this reason, the application of heuristics is encouraged. Chen and Lau [6] proposed two heuristics, namely GE and GRE, for the test suite minimization problem inspired on a previous heuristic from Harrold et al. [19]. In their experiments comparing these three heuristics, they observed that none of them was always the best. For our instantiation example and for our exploratory study we arbitrarily adopted the GE heuristic.

4.3.2. The scope-aided approach applied to a minimization example

As above stated, for this instantiation we adopted the GE heuristic from Chen and Lau [6]. The GE heuristic, depicted in Algorithm 4, is built upon a greedy algorithm based on the notions of *essential* test cases. A test case is said to be essential if it is the only test case that can cover a given entity e_i. Algorithm 4 first selects all essential test cases (first `while` loop starting at line 2); then, it repeatedly selects the test case that covers the maximum number of uncovered entities until all entities are covered (second `while` loop, line 7). In the original implementation, if there is a tie between several test cases, an arbitrary choice is made. Similar to previous examples, in this work our implementation of *getNextTestCase* (line 8) sorts the tied test cases according to their IDs to guarantee a deterministic output.

When applied to the motivating example, Algorithm 4 receives the list of entities that need to be covered (1, 2, . . . , 11, 12) and identifies the essential test cases. The first essential test case is TC_2 as it is the only one that covers statement 3. TC_2 is selected and the list of uncovered entities is updated (2, 7, 10, 11, 12). The next essential test case selected is TC_1 as it is the only one that covers statement 7. After TC_1 is selected, the list of yet to be covered entities is (10, 11, and 12) because it covered two previously uncovered statements (2 and 7). This process continues with the further selection of TC_4 (as being an essential test case to cover statement 11) and then TC_5 (an essential test case to cover statement 12). After TC_5 is selected, all the entities have been covered and Algorithm 4 returns the minimized test suite TS_m = [TC_2,TC_1,TC_4,TC_5].

Algorithm 4: Minimization (adopting the GE heuristic proposed in [6])

```
Input: T                 /*the test suite from which test cases can be selected*/
       entities                               /*list of entities to be covered*/
       coverageInfo         /*list of entities covered by each test from T*/
Output: T'         /*a subset of T, with which to test the target program*/
1  $T' \longleftarrow$ [ ]                              /*T' is initialized as an empty list*/
2  while thereAreEssentialTestCases(T, entities, coverageInfo) do
                                      /*essential test cases are selected first*/
3      add(essentialTestCase, T')
4      updateUncoveredEntities(entities, essentialTestCase)          /*removes the
        entities covered by the essential test case from the list of uncovered
        entities*/
   end
5  if isEmpty(entities) then
6      return T'
   else
7      while thereAreUncoveredEntities(T, entities, coverageInfo) do
8          selectedTestCase $\longleftarrow$ getNextTestCase(T, entities, coverageInfo)
            /*selects the test case that covers the highest number of uncovered
            entities*/
9          add(selectedTestCase, T')
10         updateUncoveredEntities(entities, selectedTestCase)       /*removes the
            entities covered by the selected test case from the list of
            uncovered entities*/
       end
   end
11 return T'
```

Notice that, for this small example, all the test cases happened to be selected as being *essential*, but this is not always the case.

When supported by the scope-aided approach, Algorithm 4 receives the list of the in-scope entities only. This is the only difference with respect to the traditional approach for minimizing test suites using the GE heuristic. The first test case selected by the scope-aided minimization is TC_1 because it is an essential test case to cover statement 7. After TC_1 is selected, there are no more essential test cases and the algorithm continues using the greedy additional approach. At this point, the entities that still need to be covered are (8 and 9) and TC_2 and TC_3 are tied as both cover these two entities. The tie is solved with TC_2 being selected and the minimization stops as all the in-scope entities have been covered. The scope-aided minimized test suite is then provided $TS_{m'}$ = [TC_1,TC_2].

5. Exploratory Study Settings

We conducted an exploratory study to assess the impact of our approach to test case prioritization, selection, and test suite minimization. In this section we discuss the settings of the study. More precisely, we focus on three research questions:

RQ1: *Scope-aided usefulness for test case prioritization*: does scope-aided approach improve test case prioritization? In particular, we will compare scope-aided prioritization with the original one (not scope-aided) with respect to fault detection rate when considering *in-scope faults* (RQ1.1) and *all faults* (RQ1.2);

RQ2: *Scope-aided usefulness for test case selection*: does scope-aided approach improve test case selection? In particular, we will compare scope-aided selection with the original one (not scope-aided) concerning the size of the selected test suite (RQ2.1) and the impact of the selection on the test suite's fault detection ability (RQ2.2);

RQ3: *Scope-aided usefulness for test suite minimization*: does scope-aided approach improve test suite minimization? In particular, we will compare scope-aided minimization with the original one (not scope-aided) with respect to the effectiveness of the minimization (RQ3.1) and the impact of the minimization on the test suite's fault detection ability (RQ3.2).

5.1. Study Subjects

In order to carry out our exploratory study and to investigate our research questions in a realistic setting, we looked for subjects in the Software-artifact Infrastructure Repository (SIR) [8]. SIR contains a set of real, non-trivial programs that have been extensively used in previous research. For selecting our subjects, some prerequisites had to be considered: first, the subjects should be written in the C language; second, they should contain faults (either real faults or seeded ones) and a test suite associated with them.

For this study we selected a total of 17 variant versions from three C subjects: grep, gzip, and sed. grep is a command-line utility that searches for lines matching a given regular expression in the provided file(s); gzip is a software application used for file compression and decompression; and sed is a stream editor that performs basic text transformations on an input stream. grep and gzip are available from SIR with 6 sequential versions (1 baseline version and 5 variant versions with seeded faults) whereas sed contains 8 sequential versions (1 baseline version and 7 variant versions with seeded faults).

Other materials were used during this study: KLEE was used in one of the steps of our approach to determine the set of in-scope entities; gcov[2] and lcov[3]

[2]`https://gcc.gnu.org/onlinedocs/gcc/Gcov.html`
[3]`http://ltp.sourceforge.net/coverage/lcov.php`

utilities were used for collecting accurate coverage metrics. MILU [22] was used to generate mutated versions of our study subjects. Finally, our own code was used to automate the majority of the steps followed during this study.

Table 2 displays additional details about our study subjects. Column "LoC" shows the lines of code[4] of each variant version. The 4th and 5th columns display the number of test cases available in the SIR test suite and the number of mutant versions created for each study subject, respectively. Please ignore for the moment the last three columns, whose meaning is explained later on.

Table 2: Details about the study subjects considered in our investigations

Sub.	**Ver.**	**LoC**	**Test Suite**	**Mutant versions**	**Mutants compiled**	**Mutants killed by SIR suite**	**Hard to kill mutants**
grep	v1	9463	199	7981	2350	325	288
grep	v2	9987	199	9471	2373	311	283
grep	v3	10124	199	9690	2425	305	272
grep	v4	10143	199	9821	2602	354	260
grep	v5	10072	199	9797	2603	374	271
gzip	v1	4594	195	5049	3965	580	354
gzip	v2	5083	195	6088	4742	499	333
gzip	v3	5095	195	4755	4261	547	249
gzip	v4	5233	195	4641	4119	593	362
gzip	v5	5745	195	5843	5072	502	335
sed	v1	5486	360	6548	1991	815	673
sed	v2	9867	360	3946	3055	919	793
sed	v3	7146	360	4125	1176	864	782
sed	v4	7086	363	7697	1838	900	815
sed	v5	13398	370	1967	1483	1000	914
sed	v6	13413	370	1889	1398	1000	925
sed	v7	14456	370	2151	1133	1000	919
	Total:	146391	4523	101459	46586	10888	8828

For carrying out our study, we needed a way of identifying different testing scopes for the investigated subjects in order to evaluate the effectiveness of our scope-aided approach. The usage scenarios depicted in the next three sections represent the testing scopes for our study.

5.1.1. Testing scopes for grep

Because of its inherent characteristics, grep made this task of identifying the testing scope relatively easy as it already considers three major usage scenarios:

1. grep -G is the *default* behavior and it interprets the provided pattern as a basic regular expression.

[4]Collected using the CLOC utility (`http://cloc.sourceforge.net/`).

2. grep -E switches grep into a special mode in which expressions are evaluated as extended regular expressions as opposed to its normal pattern matching. The essential difference between basic and extended regular expression is that some characters (e.g., '?', '+', '|', etc) have a special meaning when used in the extended mode whereas they are considered ordinary characters when used in the basic expression.
3. grep -F makes grep interpret the pattern provided as a list of fixed strings, separated by new lines, and performs an *OR* search without doing any special pattern matching.

Indeed, these scenarios are so commonly used that there are even shortcuts to them: *egrep* is equivalent to grep -E while *fgrep* corresponds to grep -F.

5.1.2. Testing scopes for gzip

For gzip, we defined the three following scenarios in which our subject could be used:

1. gzip is used, within a bigger system, for *compressing* files only;
2. gzip is used by an online service only for *decompressing* the files submitted by the service's users; and
3. gzip is used for *compressing* not only files but also whole directories *recursively.*

5.1.3. Testing scopes for sed

For sed, two usage scenarios were defined:

1. sed is used to perform basic text transformations in the contents provided in the standard input (i.e., sed scripts and input text are both provided in the standard input);
2. sed programs (or sed scripts) are used to parse the text from the input files provided.

5.2. Tasks and Procedures

In this study we considered three types of entities: function, statement, and branch, which correspondingly identify three coverage criteria. Then, for each version of the subjects investigated and for each type of entity we performed the following tasks:

1. Applied traditional prioritization, selection, and minimization techniques on the object's test suite
2. Applied our scope-aided prioritization, scope-aided selection, and scope-aided minimization on top of the traditional techniques
3. Evaluated, for each possible combination of testing scope and adequacy criteria, the performance of the scope-aided approach when compared to the original techniques and considering mutant faults.

As stated in step 3, in this study we considered mutant faults. For generating mutated versions of our study subjects we used MILU [22], a C mutation testing tool designed for both first order and higher order mutation testing that supports many mutant operators (e.g., statement deletion, constraints replacement, etc). For our study, we allowed MILU to generate first order mutants of our subjects using any of the available mutant operators.

In sections 6, 7, and 8, we report our results with respect to the set of *all faults* and *in-scope faults*. The identification of these sets is detailed next.

After the mutants are generated, we run the subject's test suite against the mutated versions to identify which mutants would be killed by the baseline test suite. These compose the set of *all faults* (mutants) considered for the next steps of our study. The number of *all faults* is displayed in the column "Mutants killed by SIR suite" in Table 2.

The ideal way to decide whether a fault is relevant or not in a given scope would be to refer to failure reports from the field where the program is used under the scope constraints. We did not have such data for our subjects, so in the aim of having unbiased data, we created, for each variant version investigated and for each specific scope, one random test suite containing 1K test cases: for each scope, the set of *in-scope faults* is composed by those mutants that are killed by the random test suite targeting that specific scope.

For generating the random test cases we developed our own scripts following a strategy very similar to that of Balcer et al. [2] for test scripts generation based on the test specification language (TSL).

We start by identifying all the possible input parameters and environment flags that can influence the behavior of the study subject (e.g., for gzip, the `-d` and `--decompress` flags can be used for decompressing files; the `-S` and `--sufix` flags can be used to define the suffix to be used for compressed files; and so on). We then investigated which input data, if any, was required by the subject (e.g., gzip can receive, as input data, a file to be either compressed or decompressed) and created the necessary support files (for gzip we created directories containing multiple files to be compressed; multiple compressed files to be tested or decompressed; etc).

Our script creates random test cases by choosing arbitrary input variables and input data to be used by the subject. To make sure that the subject's behavior is explored to its maximum, we allow our script to generate test cases that exercise unexpected behavior and error inputs (such as trying to decompress a file that is not compressed; or trying to compress a non-existent file, for example). A random test case is accepted in the random test suite for a given scope if the input domain constraints of the scope hold (e.g., for scope 2 of gzip, a valid test case should always contain the `-d`, or `--decompress`, flag).

5.3. Metrics

In this section we introduce the metrics used in our exploratory study to assess the usefulness of our approach. The *APFD* metric is used to assess scope-aided prioritization whereas *test suite size* and *impact on fault detection capability* are used to evaluate scope-aided selection and minimization.

5.3.1. Average Percentage of Faults Detected (APFD)

In order to address the research questions concerning prioritization, we need a way to assess and compare scope-aided prioritization with other prioritization techniques. In this work, we adopt the APFD metric. APFD was first introduced in [43] and in its first definition it assumed all the test costs and faults severities to be uniform. Later, a variant version of this metric incorporating varying test costs and different fault severities, the APFD$_C$, was introduced [10].

Because in our approach we consider that faults have different relevance within a given testing scope, our studies are naturally suited to the cost-cognizant version of the metric.

APFD$_C$ is calculated according to Equation 1. In this equation, T is a test suite containing n test cases with costs t_1, t_2, ..., t_n; F is a set of m faults revealed by T with severities f_1, f_2, ..., f_m; and TF_i is the first test case of an ordering T' of T that reveals fault i.

$$\text{APFD}c = \frac{\sum_{i=1}^{m}(f_i \times (\sum_{j=TFi}^{n} t_j - \frac{1}{2} t_{TFi}))}{\sum_{i=1}^{n} t_i \times \sum_{i=1}^{m} f_i} \tag{1}$$

In this work, we consider different values of severity according to fault relevance, since we focus in assessing the contribution of scope-aided prioritization to increase the speed at which in-scope faults are revealed. On the other hand, for simplicity, we consider all the tests to have the same cost, since we are not focusing on the cost of test cases.

5.3.2. Test Suite Reduction

The test suite reduction ratio achieved by test case selection or test suite minimization is measured according to Equation 2.

$$\text{Reduction} = \left(1 - \frac{\#\ \textit{test cases in the reduced test suite}}{\#\ \textit{test cases in the original test suite}}\right) \times 100\% \tag{2}$$

We apply this formula both to scope-aided selection and minimization approaches, and to the original (not scope-aided) ones.

5.3.3. Impact on Fault Detection Capability

The impact on fault detection capability of a given test suite is calculated according to Equation 3.

$$\text{Impact} = \left(1 - \frac{\#\ \textit{faults detected by the reduced test suite}}{\#\ \textit{faults detected by the original test suite}}\right) \times 100\% \tag{3}$$

As above, we apply this formula both to scope-aided selection and minimization approaches, and to the original (not scope-aided) ones.

5.4. Execution

Many of the mutants generated by Milu [22] would not compile and this represented a first filtering to reduce the amount of computation required to define the sets of *all faults* and *in-scope faults* (the number of mutants that compiled for each study subject is displayed in the 6th column of Table 2). However, for some subjects, the number of mutants that compiled was still too much. For gzip, for example, the average number of compiled mutants for each version was 4431. We then decided that, for each variant version investigated, we would run the baseline test suite (the one from SIR) on a set of 1K randomly selected mutants.

When evaluating the test suites, we noticed that one of the test cases available for gzip requires the compression of a 2GB file and, even after our decision of selecting "only" 1K mutants per version, we would still need to run that test case 5k times. For this reason, we decide to remove this test cases and a few others that would add a lot of computation overhead in our already expensive study, without contributing with significant added value. The final test suite for gzip considered in our study consisted of 195 test cases.

For running the random test suites to identify the set of in-scope faults, we are interested only in those mutants that could be killed by the baseline suite (column "Mutants killed by SIR suite" of Table 2). However, after a preliminary analysis of this set, we noticed that for some subjects, in particular for gzip, many mutants could be killed by the vast majority of the test cases, and a considerably amount of them, even by any test case. Because in our study we want to evaluate the ability of the scope-aided approach to identify the most relevant faults to a given testing scope, we followed the suggestions given in [1] and decided to eliminate the easy-to-kill mutants before proceeding to the next steps. We considered only the mutants that could not be detected by more than 50% of the test cases (these are displayed in the last column of Table 2). After this filtering, we proceeded with the execution of the random test suites.

As the reader may have noticed already, these steps required a lot of computational effort. About 4.5 million tests were run over the mutated versions only for the identification of the set of *all faults* (199 tests from grep $\times$ 5000 mutants + 195 tests from gzip $\times$ 5000 mutants + $\sim$ 365 tests from sed $\times$ 7000 mutants). After this step, we still needed to run the random test suites (3k tests for grep and gzip, and 2k tests for sed) over the set of hard-to-kill mutants. Obviously, however, these steps are not required for the adoption of our approach as they were applied only for supporting our empirical studies.

6. Prioritization Study (RQ1)

As previously stated, prioritization aims at reordering a test suite so that potential faults are revealed faster. A natural question that arises, thus, is whether scope-aided prioritization helps traditional approaches to provide faster detection of faults in reuse testing. In this section we investigate the usefulness of scope-aided approach to support the test case prioritization task with respect to the following research questions:

RQ1.1: *In-scope faults detection rate*: how does scope-aided prioritization compare with original (not scope-aided) prioritization with respect to fault detection rate when considering *in-scope faults*?

RQ1.2: *All faults detection rate*: how does scope-aided prioritization compare with original (not scope-aided) prioritization with respect to fault detection rate when considering *all faults*?

We answer these questions by comparing, for each prioritization approach considered in this study, the original approach with the scope-aided one in terms of the weighted average of the percentage of faults detected, or APFD_C, over the life of a given test suite.

With the purpose of having a broader view of the extent to which the scope-aided approach could impact the traditional prioritization techniques, we wanted to apply it to different approaches to prioritization: we focused on coverage-, similarity-, and search-based strategies, which represent most of the used approaches. To select the specific techniques among those found in the literature, we used three criteria: the technique should be well documented so to be properly reproduced; it should be proven to be effective; and, given the size of the studies, it should be computationally affordable. In the end, we selected two coverage-based prioritization approaches (the well-known *total* and *additional* greedy heuristics); one similarity-based approach (proposed by Jiang et al. [23]); and one search-based technique (one instantiation of the Hill Climbing algorithm applied to test prioritization as proposed in the work from Li et al. [29]).

6.1. RQ1.1: In-scope faults detection rate

To answer RQ1.1, we assess scope-aided prioritization in comparison with the traditional techniques when considering the in-scope faults. For calculating the APFD_C values, we assign 0 as the severity of the out-of-scope faults to consider only the in-scope ones, getting severity equal to 1 ($\text{APFD}_C(1{,}0)$).

All the results are reported in the Figures 1, 2 and 3, for the three coverage criteria considered.

Considering the overall average for each subject, the scope-aided approach improved the APFD_C for all the subjects when considering the coverage-based approaches, with the biggest improvement being an increase of 40.35% in the average APFD_C for gzip using the total approach (from 49.05 to 68.84). The smallest improvement happened for gzip using the additional approach, but in that case the scope-aided approach achieved only a negligible improvement of less than 1% in the overall average APFD_C. Concerning the similarity-based approach, scope-aided improved the APFD_C for grep and gzip, and it was almost tied with the traditional technique for sed (overall average APFD_C of 93.46 for the traditional technique; and 93.29 for the scope-aided one). The biggest increase in the overall average APFD_C was a 10.12% improvement for gzip. With respect to the search-based approach, scope-aided was defeated by the traditional technique when considering gzip; it was basically tied when considering sed (overall average APFD_C of 95.83 for the traditional technique;

and 95.82 for the scope-aided one); and it slightly improved (1.33%) the overall average APFD$_C$ for grep.

Figure 1 displays the impact of the adoption of scope-aided approach with respect to the *Function* coverage criterion. Each subfigure displays the consolidated results related to a given approach (either total, additional, similarity, or search-based) applied to one of the subjects investigated in our study (grep, gzip, or sed). The x-axes represent the versions of each subject (recall that we analyzed all the versions of our study subjects available from SIR, i.e., 5 versions for grep and gzip, and 7 versions for sed), while the y-axes display the impact on the APFD$_C$ metric. The height of the vertical bars represents the contribution achieved by the adoption of the scope-aided approach for each testing scope. In particular, for similarity and search-based the contribution is obtained as an average of 50 runs, to account for randomness in these techniques.

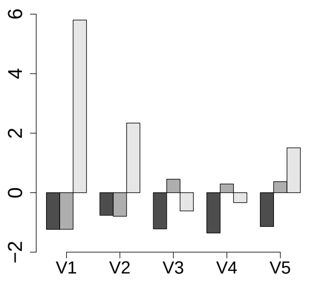
For inter-graph comparison, observe that (besides the bar height) the values of the y-axes should also be considered as the scale of that axis changes across the graphs. Let us consider, for example, Figures 1b and 1d: even though the biggest bars in each graph have more or less the same size, they represent very different values of the contribution to the APFD$_C$ (approximately 90 in Figure 1b, and approximately 2 in Figure 1d).

For grep and gzip the leftmost bar (dark grey) is related to scope 1; the central bar (grey) is associated with scope 2; and the rightmost bar (light grey) represents the scope 3. For sed, because we investigated two testing scopes, we have only two bars for each version: scope 1, represented by the leftmost bar (dark grey); and scope 2, associated with the rightmost bar (light grey). Bars with negative values mean that the adoption of scope-aided approached achieved an APFD$_C$ lower than the one achieved by the traditional prioritization approach (not scope-aided). If a bar is not visible it is because, for that case, our approach was tied with the traditional technique.

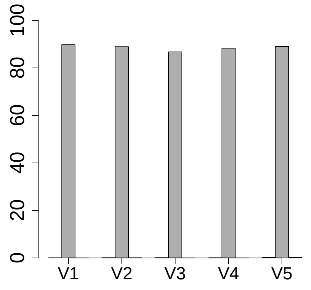
Figures 2 and 3 are analogous to Figure 1 and they display the consolidated data with respect to the *Statement* and *Branch* coverage criteria, respectively.

As regards the function coverage criterion (Figure 1), the scope-aided approach improved the original APFD$_C$ in 91 cases; it was tied with the traditional technique in 12 cases; and it was defeated 73 times. The highest improvement was achieved for gzip v1 when adopting the total prioritization approach for the scope 2 (Figure 1b). The scope-aided approach obtained an APFD$_C$ of 98.57 against 8.78 achieved by the traditional technique. The greatest loss, on the other hand, occurred for gzip v4 when applying the search-based prioritization for the scope 1 (Figure 1k); the traditional approach achieved an APFD$_C$ of 80.13 and the scope-aided one achieved 73.86. Overall, the average APFD$_C$ achieved by scope-aided prioritization (88.84) was higher than the one achieved by the original techniques (86.36).

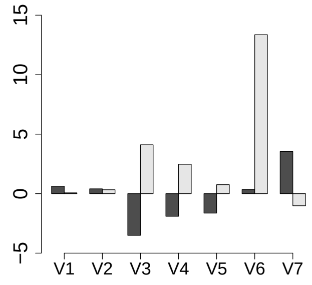
For the statement coverage criterion (Figure 2), once again the scope-aided approach performed better than the traditional one in the majority of the cases (101 times against 75); it also achieved a better average APFD$_C$ (88.55) than the traditional approaches (86.82). This time, both the best and the worst results were associated with the same combination of subject and prioritization



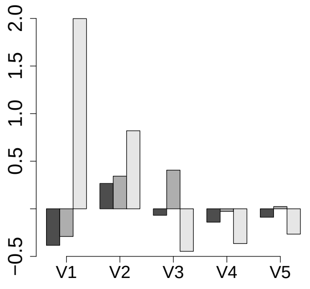
(a) **Total** grep



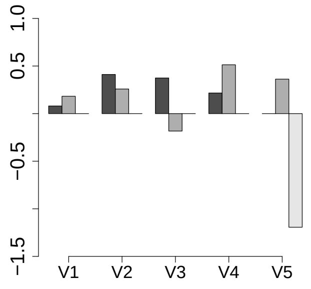
(b) **Total** gzip



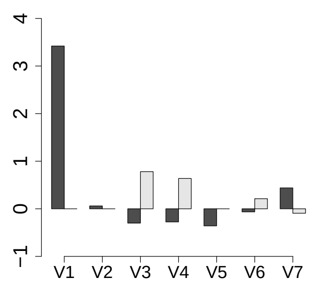
(c) **Total** sed



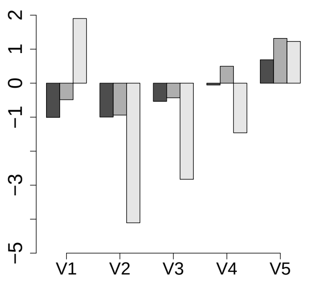
(d) **Additional** grep



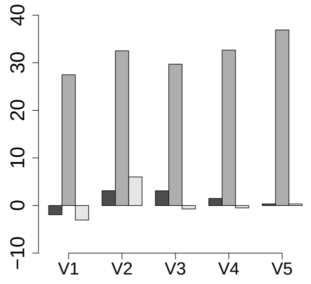
(e) **Additional** gzip



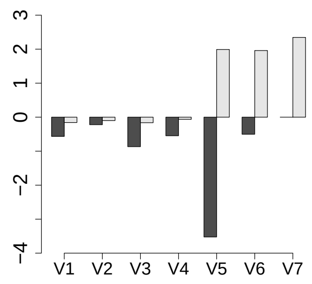
(f) **Additional** sed



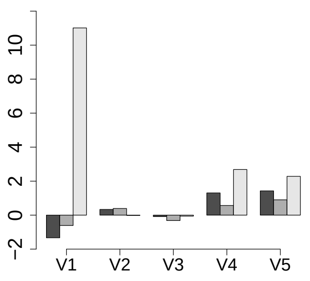
(g) **Similarity** grep



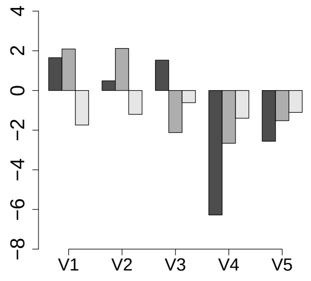
(h) **Similarity** gzip



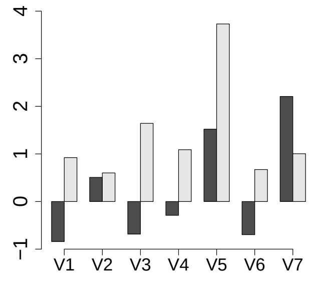
(i) **Similarity** sed



(j) **Search-based** grep

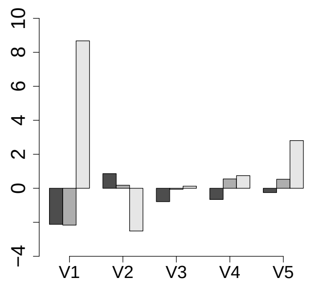


(k) **Search-based** gzip

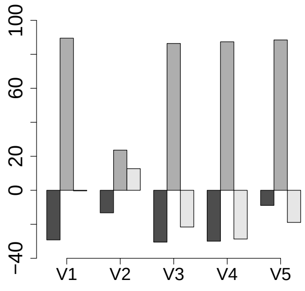


(l) **Search-based** sed

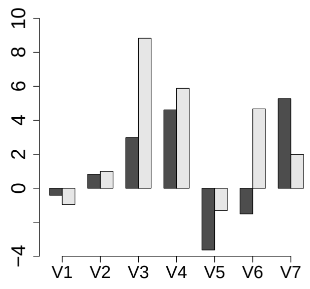
Figure 1: Contribution to the APFD$_C$ when considering the set of ***in-scope faults*** and the ***Function*** coverage criterion



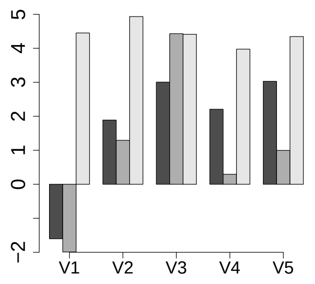
(a) **Total** grep



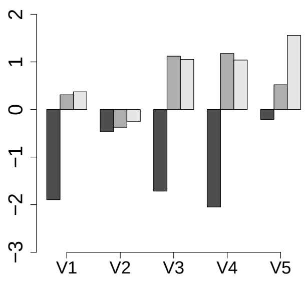
(b) **Total** gzip



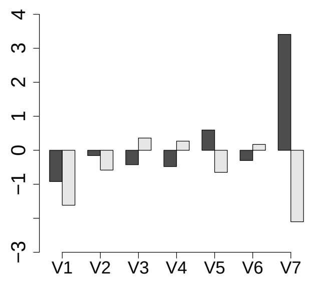
(c) **Total** sed



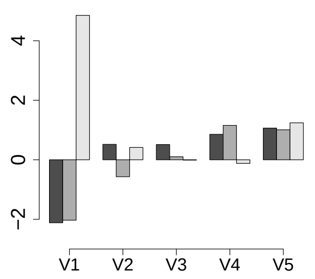
(d) **Additional** grep



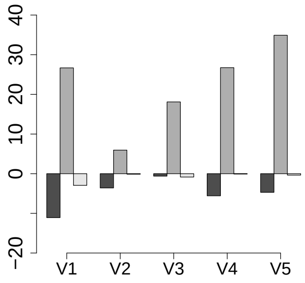
(e) **Additional** gzip



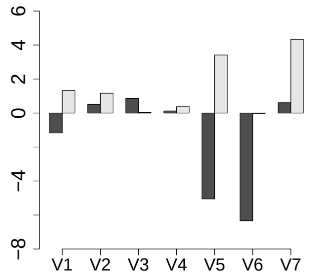
(f) **Additional** sed



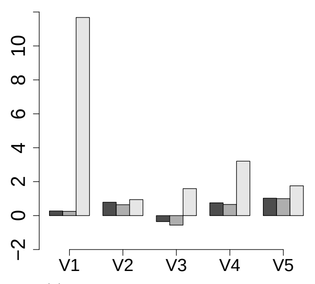
(g) **Similarity** grep



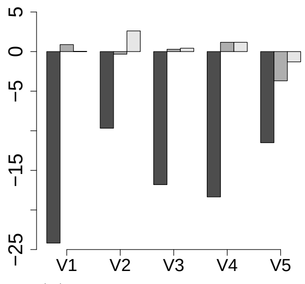
(h) **Similarity** gzip



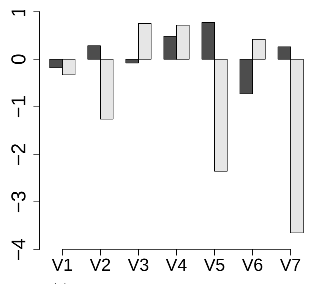
(i) **Similarity** sed



(j) **Search-based** grep



(k) **Search-based** gzip



(l) **Search-based** sed

Figure 2: Contribution to the APFD$_C$ when considering the set of ***in-scope faults*** and the ***Statement*** coverage criterion

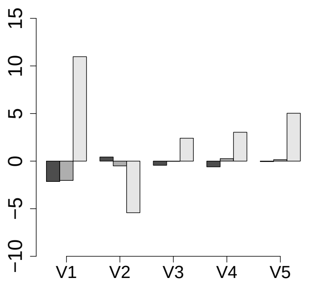
strategy. The greatest loss was noticed for gzip v3 with the total algorithm for scope 1 (46.31 against 76.80), whereas the highest improvement was seen for the version 1 and scope 2 of gzip (Figure 2b). The scope-aided test suite yielded an APFD$_C$ of 98.71 against 9.21 achieved by the traditional technique.

The scope-aided approaches also outperformed the traditional ones when looking at the overall results from the perspective of the branch coverage criterion (Figure 3). The results were similar to the ones obtained for the statement coverage criterion and, again, the best and worst results were associated with the total algorithm being applied for prioritizing the test suites from gzip. The highest improvement in the APFD$_C$ was observed for the version 1 and testing scope 2 (improved from 9.86 to 98.63); whereas the worst result was seen for the version 1 and scope 2 (26.09 against 75.13). All over, the scope-aided approach improved the original APFD$_C$ 94 times; it was tied with the traditional technique in 1 single case; and it was defeated 81 times. Besides that, it achieved a higher average APFD$_C$ value (88.71 against 86.72 achieved by the traditional techniques).

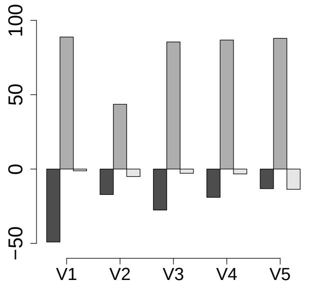
One consistent result across all the coverage criteria was the fact that the biggest discrepancy between results was always related to the total approach being applied for the gzip subject. One possible explanation for that is the fact that the test suite for gzip contains many test cases that are very similar to each other. Many test cases related to the compression task, for example, are repeated different times for each different compression level supported (from 1 to 9), which in the end may have a very small impact in the set of entities exercised by those test cases (or may even not impact it at all). Recall that the total strategy will prioritize the test cases according to the total number of entities covered by them and test cases that are very similar to each other will probably cover the same amount of entities. Besides that, there is a big chance that very similar test cases will reveal the same set of faults. When the total approach puts these "groups" of test cases all together, two main things could happen that may have a big impact in the APFD$_C$: either all test cases in a given group reveal many faults (in such case, the first test case in the group will contribute immediately to the increase of the APFD$_C$) or none of the cases in the group reveals any fault (this would *freeze* the contribution to the APFD$_C$ metric until all test case in that group are evaluated).

We manually evaluated the set of in-scope faults assigned for each scope and for all the versions of gzip and confirmed that specially for the scope 2, we had a situation in which a few test cases would reveal a big set of faults whereas the vast majority of the test cases would either reveal just a small set of faults or not reveal any fault at all. In such scenario, a good (or bad) choice of the first test case can have a huge impact in the APFD$_C$ achieved by the prioritized test suite.

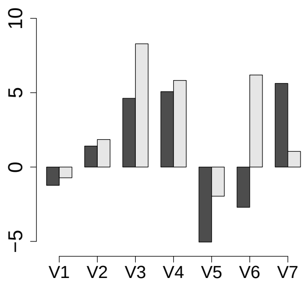
Because test case prioritization seeks to order the test cases in a way to maximize the benefits even if the testing needs to be prematurely halted at some point, we also investigated the results achieved by the scope-aided prioritization when considering different fractions of a given test suite (75%, 50%, and 25%).



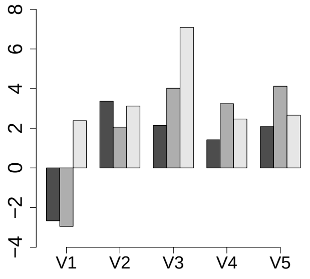
(a) **Total** grep



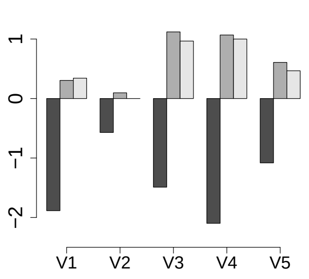
(b) **Total** gzip



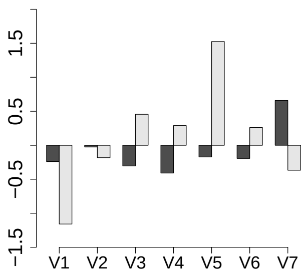
(c) **Total** sed



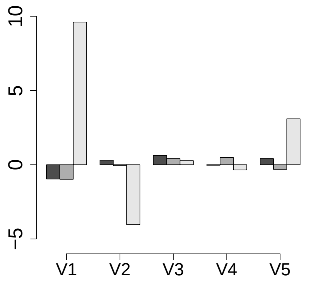
(d) **Additional** grep



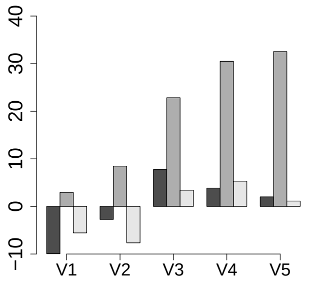
(e) **Additional** gzip



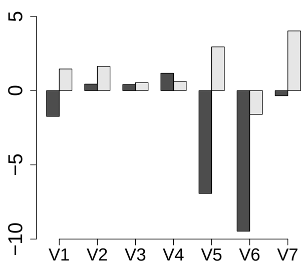
(f) **Additional** sed



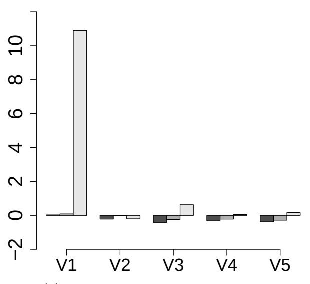
(g) **Similarity** grep



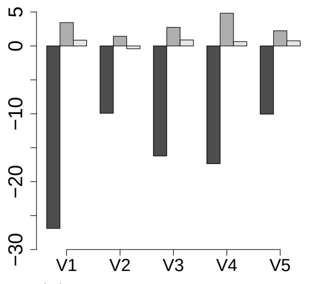
(h) **Similarity** gzip



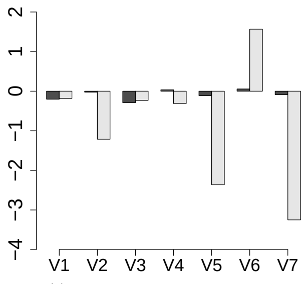
(i) **Similarity** sed



(j) **Search-based** grep



(k) **Search-based** gzip



(l) **Search-based** sed

Figure 3: Contribution to the APFD$_C$ when considering the set of ***in-scope faults*** and the ***Branch*** coverage criterion

Table 3: Average APFD$_C$(1,0) (and coefficient of variation) when considering different fractions of the prioritized suites

Coverage criterion	**Fraction: 75%**		**Fraction: 50%**		**Fraction: 25%**	
	original	scope-aided	original	scope-aided	original	scope-aided
Function	88.0 (0.13)	**88.6** (0.11)	87.6 (0.13)	**88.2** (0.10)	85.0 (0.13)	**85.1** (0.10)
Statement	88.9 (0.11)	88.8 (0.12)	86.2 (0.13)	**87.9** (0.10)	84.4 (0.13)	**86.2** (0.09)
Branch	89.0 (0.10)	88.3 (0.13)	87.9 (0.10)	87.3 (0.12)	85.1 (0.10)	**86.4** (0.09)
Average:	88.6	88.6	87.2	**87.8**	84.8	**85.9**

The results obtained are displayed in Table 3. The values provided are grouped by coverage criterion and they represent the average APFD$_C$ achieved for the different fractions of the prioritized test suites. The values enclosed in parentheses represent the coefficient of variation[5]. We highlight in bold the cases in which scope-aided prioritization performed better than the original (not scope-aided) prioritization.

Overall, scope-aided prioritization performed better than the original one in the majority of the cases, except when considering the fractions 50% and 75% for branch and the fraction 75% for statement where not scope-aided performed better.

Even if our approach outperformed the traditional ones in the majority of the cases, in Table 3 we can only see small improvements in the average APFD$_C$. It is important to observe though that for our experiments we adopted state-of-the-art prioritization approaches that already achieved very high APFD$_C$ values when applied in the traditional way. Thus, the scope-aided boost could improve on top of approaches that were already good.

With the purpose of better understanding the contribution provided by the scope-aided approach for the prioritization task, we also evaluated the results of our study from the perspective of the different prioritization strategies investigated. The results are displayed in Table 4 and the values provided are grouped by the combination of prioritization approach and coverage criterion.

When considering the approaches total, additional, and similarity, scope-aided prioritization performed better than the original ones for all the coverage criteria considered. Concerning the search-based approach, the results from scope-aided prioritization were better for function, and worse for statement and branch.

[5]The coefficient of variation (CV), also known as relative standard deviation (RSD), is the ratio of the standard deviation σ to the mean μ. Different from the standard deviation that must always be understood in the context of the mean, the coefficient of variation allows the comparison between data sets with different means and even different units. The lower the value of CV, the lower the variance of that data set. Looking at Table 3, for example, we can tell that the combinations statement–25% and branch–25%, both with a CV of 0.09, had the lower variance across all the possible combinations of coverage criteria and test suite fractions.

Table 4: Average APFD$_C$(1,0) (and coefficient of variation) when considering different prioritization approaches and different coverage criteria

Approach	**Function**		**Statement**		**Branch**	
	original	scope-aided	original	scope-aided	original	scope-aided
Total	77.0 (0.35)	**87.6** (0.14)	75.6 (0.34)	**81.0** (0.24)	74.2 (0.34)	**80.6** (0.24)
Additional	92.1 (0.07)	**92.3** (0.07)	94.1 (0.06)	**94.9** (0.05)	94.7 (0.05)	**95.5** (0.04)
Similarity	83.6 (0.18)	**87.3** (0.10)	86.1 (0.13)	**88.1** (0.08)	86.4 (0.12)	**88.5** (0.06)
Search-based	89.8 (0.08)	**90.2** (0.08)	91.6 (0.06)	90.2 (0.08)	91.6 (0.05)	90.2 (0.08)
Average:	85.7	**89.4**	86.8	**88.5**	86.7	**88.7**

6.2. RQ1.2: All faults detection rate

To answer RQ1.2, we assess scope-aided prioritization in comparison with traditional techniques when considering all faults. We calculated APFD$_C$ assigning the same severity for all the faults. Because we also consider all the test cases to have the same costs, the APFD$_C$ formula reduces to the traditional APFD in which costs are not considered.

Overall, the results achieved by the scope-aided approach when considering *all faults* were slightly worse than the ones achieved when considering only the in-scope faults, but still very competitive with the traditional techniques. This was a positive result to us: we were somewhat prepared to see a greater lose in the effectiveness when considering all the faults given that our approach targets the in-scope ones.

For the sake of space, we do not replicate the barplots that would look similar to the ones provided for RQ1.1. Instead, we highlight below the main results.

When considering the average among all the subjects, the average APFD$_C$ achieved by scope-aided (87.25) was better than the one achieved by the traditional approaches (86.54), but to a smaller extent when compared with the results achieved in the study considering in-scope faults. The biggest improvement observed was an increase of 16.5% in the average APFD$_C$ for gzip using the total approach (from 57.76 to 67.29), whereas the biggest defeat was a reduction of 4.74% in the average APFD$_C$ for gzip using the search-based approach.

Table 5 displays the average APFD$_C$ achieved by the scope-aided and the traditional approaches grouped by prioritization strategy and coverage criterion. Scope-aided prioritization performed always better than the corresponding original approach when considering the total strategy; and it performed always worse when considering the additional strategy; for similarity and search-based strategies the results varied. Even though the traditional approaches defeated the scope-aided ones in a bigger number of cases, scope-aided still achieved better results when considering the overall average.

Table 5: Average APFD$_C$(1,1) (and coefficient of variation) when considering different prioritization approaches and different coverage criteria

Approach	**Function**		**Statement**		**Branch**	
	original	scope-aided	original	scope-aided	original	scope-aided
Total	74.4 (0.24)	**78.4** (0.23)	73.8 (0.23)	**76.0** (0.26)	70.6 (0.31)	**74.7** (0.30)
Additional	92.9 (0.07)	92.6 (0.07)	95.5 (0.05)	95.4 (0.04)	96.2 (0.04)	95.9 (0.03)
Similarity	84.3 (0.11)	**86.5** (0.09)	85.9 (0.08)	85.7 (0.10)	87.0 (0.06)	86.5 (0.10)
Search-based	91.5 (0.04)	**91.6** (0.04)	93.1 (0.04)	91.6 (0.04)	92.8 (0.03)	91.5 (0.03)
Average:	85.8	**87.3**	87.1	**87.2**	86.6	**87.1**

7. Selection Study (RQ2)

In this section we investigate the usefulness of scope-aided approach to support the test case selection task with respect to the following research questions:

RQ2.1: *Test suite reduction*: how does scope-aided selection compare with the original one (not scope-aided) in terms of test suite reduction achieved?

RQ2.2: *Impact on fault detection capability*: what is the impact of scope-aided selection with respect to the test suite's fault detection capability when compared to the original (not scope-aided) selection and considering both *all faults* and *in-scope faults*?

7.1. RQ2.1: Test suite reduction

To answer RQ2.1, we assess scope-aided selection in comparison with the traditional technique in terms of the test suite reduction rate achieved by each approach. The reduction rate is calculated according to Equation 2 (detailed in Section 5.3.2).

Table 6 displays the average reduction achieved and the results are grouped by the different versions of each subject evaluated in our study. Scope-aided outperformed the traditional approach in all the cases considered with an average extra reduction of 10.69% for grep, 5.60% for gzip, and 3.94% for sed.

The smallest reduction achieved by the traditional selection was 65.33% and the biggest one was 98.61%; for scope-aided selection, the smallest reduction was 74.37% and the biggest one was 99.72%. In both cases, the smallest reduction was associated with grep while targeting the branch coverage criterion, and the biggest reduction was related to sed while targeting the function coverage criterion.

When looking from the perspective of the different coverage criteria considered, the smallest rates of reduction achieved were 93.85% for the traditional approach and 95.98% for the scope-aided one (for function); 72.86% for the traditional approach and 78.89% for the scope-aided one (for statement); and

Table 6: Comparison between the average test suite reduction (and coefficient of variation) achieved by the scope-aided selection and the traditional approach

Subject versions	***grep***		***gzip***		***sed***	
	original	scope-aided	original	scope-aided	original	scope-aided
V1	77.7% (0.17)	**87.2%** (0.11)	91.6% (0.02)	**97.3%** (0.01)	93.6% (0.04)	**97.3%** (0.02)
V2	77.6% (0.17)	**88.7%** (0.08)	91.6% (0.02)	**97.0%** (0.02)	93.9% (0.04)	**98.1%** (0.02)
V3	77.9% (0.17)	**88.7%** (0.10)	91.8% (0.02)	**97.5%** (0.01)	93.5% (0.04)	**97.2%** (0.02)
V4	78.1% (0.16)	**89.0%** (0.09)	91.6% (0.02)	**97.3%** (0.02)	93.2% (0.04)	**97.2%** (0.02)
V5	78.1% (0.16)	**89.2%** (0.09)	91.8% (0.03)	**97.3%** (0.02)	93.3% (0.04)	**97.3%** (0.02)
V6	-	-	-	-	93.5% (0.04)	**96.6%** (0.03)
V7	-	-	-	-	92.9% (0.04)	**97.7%** (0.01)
Average:	77.9%	**88.6%**	91.7%	**97.3%**	93.4%	**97.4%**

65.33% for the traditional approach and 74.37% for the scope-aided one (for branch).

The scope-aided approach was expected to perform better in this metric because, by construction, the test case selection is targeted to a subset of the (testing) input domain. In fact, RQ2.1 was a *proof of concept* that demonstrated preliminary positive results regarding the adoption of scope-aided approach for test case selection. We now proceed to investigate what is the impact on the fault detection capability caused by the test suite reduction.

7.2. RQ2.2: Impact on fault detection capability

We answer RQ2.2 by assessing the scope-aided selection in comparison with the traditional technique in terms of the impact on fault detection capability generated by each approach. Such impact is calculated according to Equation 3 (detailed in Section 5.3.3). For this research question we considered both the set of *all faults* and the *in-scope faults*.

Figure 4 displays the boxplots for each approach grouped by the different coverage criteria considered. The y-axis displays the impact (in %) on fault detection capability. For this metric, the lower the impact, the better.

Overall, the traditional approach defeated the scope-aided one in all the cases. In average, the scope-aided generated an extra impact of 6.77%, 12.16%, and 13.84% for function, statement, and branch, respectively. Even though the differences in the average were not very big, the data was more spread for the scope-aided approach.

These results were in accordance with our initial intuition as we did not expect the scope-aided approach to generate a lower impact on the fault detection capability than the traditional one when considering all the faults.

When considering the set of in-scope faults (Figure 5), the scope-aided approach improved and it defeated the traditional one for the statement and branch coverage criteria. In average, scope-aided generated an extra impact of 3.20% for function, but it created 1.70% and 1.25% *less* impact for statement and branch, respectively.

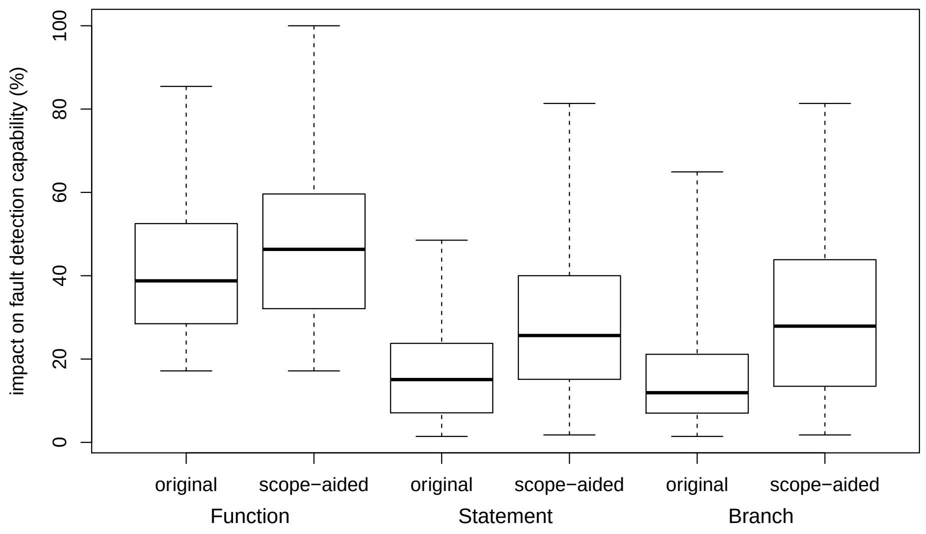


Figure 4: Impact on fault detection capability for the different coverage criteria when considering the set of *all faults*

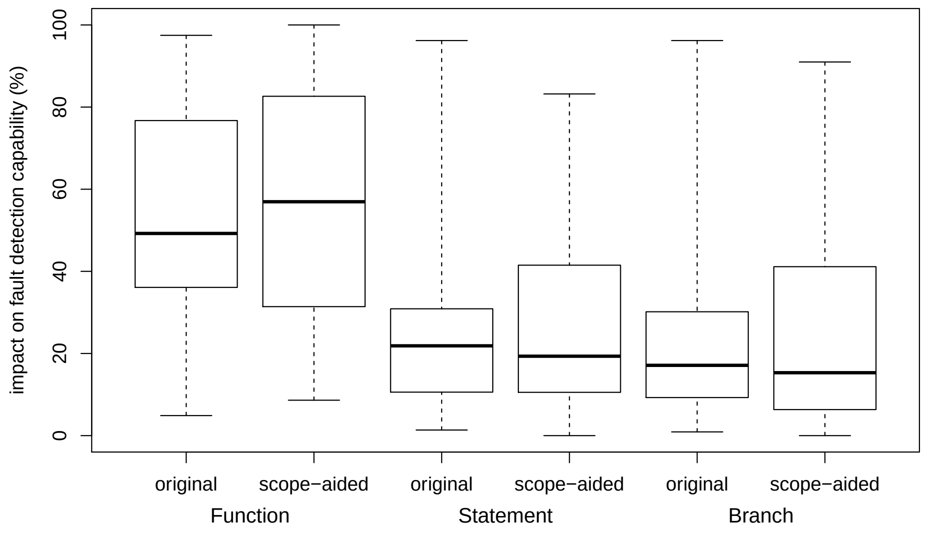


Figure 5: Impact on fault detection capability for the different coverage criteria when considering the set of *in-scope faults*

Looking from the perspective of the different subjects investigated, scope-aided approach achieved better results than the traditional one for grep and gzip with an average impact of 32.85% (against 37.30% generated by the traditional) and 35.68% (against 43.05% generated by the traditional), respectively. For sed, scope-aided generated an average impact of 39.70%, whereas the traditional approach generated an impact of 26.76%.

8. Minimization Study (RQ3)

In this section we finally investigate the usefulness of scope-aided approach to support the test case minimization task. The research questions that we address are the same as the set of RQ2.1 and RQ2.2 already considered for selection, except that here we refer to the minimization task:

RQ3.1: *Test suite reduction*: how does scope-aided minimization compare with the original one (not scope-aided) in terms of test suite reduction achieved?

RQ3.2: *Impact on fault detection capability*: what is the impact of scope-aided minimization with respect to the test suite's fault detection capability when compared to the original (not scope-aided) minimization and considering both *all faults* and *in-scope faults*?

8.1. RQ3.1: Test suite reduction

We answer RQ3.1 in an analogous way to that applied for RQ2.1 (Section 7.1) and we assess scope-aided minimization in comparison with the traditional technique in terms of the test suite reduction rate achieved by each approach.

Table 7 displays the average reduction achieved and the results are grouped by the different versions of each subject evaluated in our study. Scope-aided outperformed the traditional approach in all the cases considered with an average extra reduction of 10.53% for grep, 5.57% for gzip, and 3.69% for sed.

Table 7: Comparison between the average test suite reduction (and coefficient of variation) achieved by the scope-aided minimization and the traditional approach

Subject versions	***grep***		***gzip***		***sed***	
	original	scope-aided	original	scope-aided	original	scope-aided
V1	77.7% (0.17)	**87.4%** (0.11)	91.6% (0.02)	**97.3%** (0.01)	94.1% (0.04)	**97.5%** (0.02)
V2	77.7% (0.17)	**88.9%** (0.09)	91.8% (0.02)	**97.1%** (0.02)	93.9% (0.04)	**98.1%** (0.02)
V3	78.4% (0.16)	**88.9%** (0.10)	91.8% (0.02)	**97.5%** (0.01)	93.8% (0.03)	**97.3%** (0.02)
V4	78.6% (0.16)	**89.2%** (0.09)	91.8% (0.02)	**97.4%** (0.01)	93.4% (0.04)	**97.3%** (0.02)
V5	78.6% (0.16)	**89.3%** (0.09)	91.8% (0.03)	**97.3%** (0.02)	93.8% (0.04)	**97.2%** (0.02)
V6	-	-	-	-	93.8% (0.04)	**96.6%** (0.03)
V7	-	-	-	-	93.3% (0.04)	**97.8%** (0.01)
Average:	78.2%	**88.7%**	91.8%	**97.3%**	93.7%	**97.4%**

The smallest reduction achieved by the traditional minimization was 65.83% and the biggest one was 98.61%; for scope-aided minimization, the smallest reduction was 74.37% and the biggest one was 99.72%. In both cases, the smallest reduction was associated with grep while targeting the branch coverage criterion, and the biggest reduction was related to sed while targeting the function coverage criterion.

When looking from the perspective of the different coverage criteria considered, the smallest rates of reduction achieved were 94.36% for the traditional approach and 96.48% for the scope-aided one (for function); 72.86% for the traditional approach and 78.89% for the scope-aided one (for statement); and 65.83% for the traditional approach and 74.37% for the scope-aided one (for branch).

As previously explained in Section 7.1, the scope-aided approach was expected to perform better in this metric. We now evaluate the impact of the test suite reduction on the fault detection capability.

8.2. RQ3.2: Impact on fault detection capability

To answer RQ3.2, we assess the scope-aided minimization in comparison with the traditional technique in terms of the impact on fault detection capability generated by each approach. This question is analogous to RQ2.2 (Section 7.2) and we use the same equation (Equation 3) to calculate the impact on fault detection capability of a given test suite.

Figure 6 displays the boxplots for each approach grouped by the different coverage criteria considered. The results slightly improved with respect the ones obtained for the same metric in our study with test case selection (Figure 4, Section 7.2).

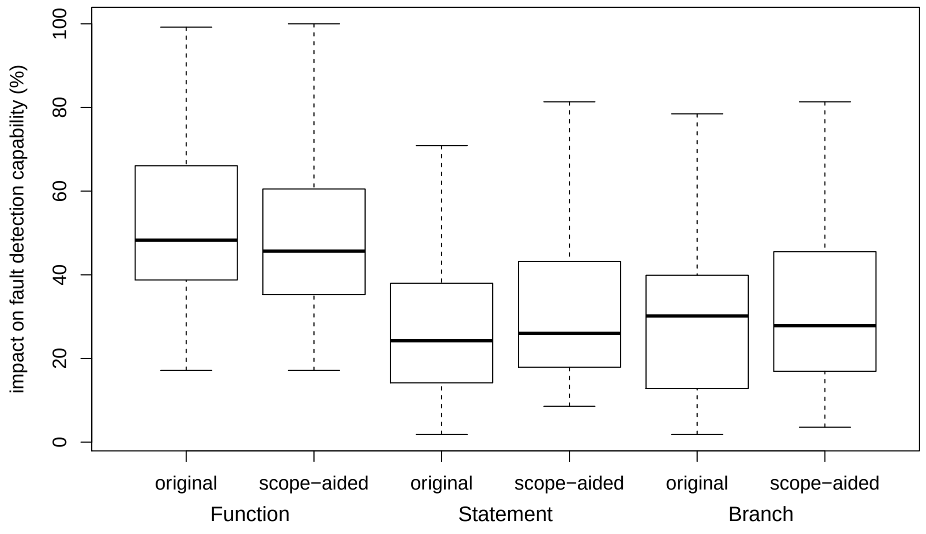


Figure 6: Impact on fault detection capability for the different coverage criteria when considering the set of *all faults*

When considering the median values, scope-aided produced better results for function and branch, and it was defeated for the statement criterion. With respect to the average, the scope-aided minimization produced better result for function and it was defeated by the traditional approach for the statement and branch coverage criteria. The traditional minimization generated an extra impact of 3.38% for function, whereas scope-aided generated 5.67% for statement and 4.74% for branch. Again, these results were in accordance with our initial intuition.

Figure 7 displays the boxplots for each approach when considering the set of in-scope faults. The results achieved by the scope-aided approach improved considerably with respect to previous comparison considering the set of all faults. Overall, the scope-aided approach defeated the traditional one in all the cases. In average, the traditional approach generated an extra impact of 6.54%, 7.75%, and 9.95% for function, statement, and branch, respectively.

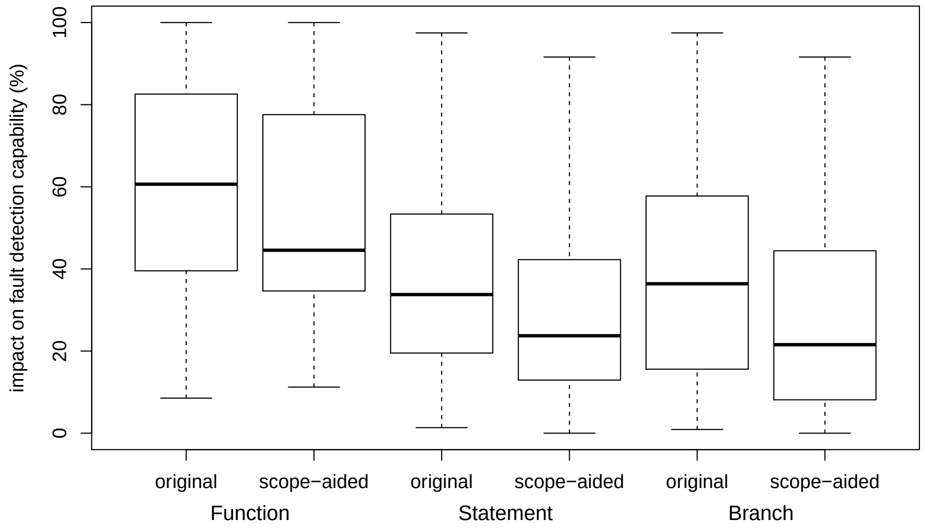


Figure 7: Impact on fault detection capability for the different coverage criteria when considering the set of *in-scope faults*

When looking from the perspective of the different subjects investigated, the results were very similar to the ones observed in our study with selection (Section 7.2): scope-aided approach achieved better results than the traditional one for grep and gzip with an average impact of 30.72% (against 33.80% generated by the traditional) and 36.16% (against 59.92% generated by the traditional), respectively. For sed, scope-aided generated an average impact of 47.62%, whereas the traditional approach generated an impact of 44.28%.

9. Discussion on Costs and Benefits of the Approach

In sections 6, 7, and 8 we investigated the potential usefulness of scope-aided test prioritization, selection, and minimization. On the other hand, we also need to consider the possible costs of our proposed approach. While only data collected from real-world usage (which are not available) could provide conclusive evidence about whether such costs are sustainable and justified, in this section we present a preliminary assessment regarding the costs incurred by the adoption of the proposed approach.

In general, because the cost-benefit ratio incurred by the adoption of a given technique largely depends on the specific testing scenario in which the technique is going to be applied, deciding whether or not its adoption is worthwhile needs to be done on a case-by-case basis. For example, in the ideal case where the test suite is fully automated and one can analyze the test results within a negligible time, any prioritization technique, even the most efficient one, would be worthless in practical terms. The same could be said for selection and minimization techniques.

These techniques become attractive when the execution of the whole test suite is not doable due to resource constraints (e.g., computer-time, person-time, etc); or, for the specific case of prioritization, when the execution of the whole test suite would take so long that it is worth prioritizing the test cases in order to reveal potential faults faster and to maximize the benefits in the case that testing needs to be prematurely halted.

The literature offers countless variations of prioritization, selection, and minimization techniques. In all cases, these techniques start from a test suite T and derive a test suite T', which is a reordering of T in case of prioritization, and a subset of T in case of selection or minimization. Let us indicate C_{ORIG} as the cost of executing the original test suite T, and C_{POST} as the cost of testing after applying the technique. C_{ORIG} can be expressed as $C_{exec} + C_{act}$, and C_{POST} as $C_{tech} + C_{exec'} + C_{act'}$, where C_{tech} is the cost (e.g., effort, time) required for applying the technique (minimizing a test suite, for example); C_{exec} and $C_{exec'}$ are the cost required for running the original test suite T and the set T' resulting from the application of the technique, respectively; and finally C_{act} and $C_{act'}$ denote the cost of analyzing the test results and taking appropriate actions (fixing the bugs founds, for example) before and after applying the technique, respectively.

We recall that our approach is proposed as a boost of other existing prioritization, selection and minimization techniques. So, by applying a scope-aided (prioritization, selection or minimization) technique we obtain a test suite T'_s different from the test suite T' that would be obtained by the same technique without considering scope. As in previous sections we assessed the benefits of our approach against some state-of-the-art prioritization, selection and minimization approaches, consistently we will discuss augmented or reduced costs of scope-aided approaches with respect to the original (not scope-aided) ones.

With regards to the cost of applying the technique (C_{tech}), when compared to the original technique, our approach has an extra cost to identify the in-scope entities, which will depend on the method chosen for performing this task. We remind that our approach consists of two main steps: (1) constraints identification and (2) in-scope entities identification.

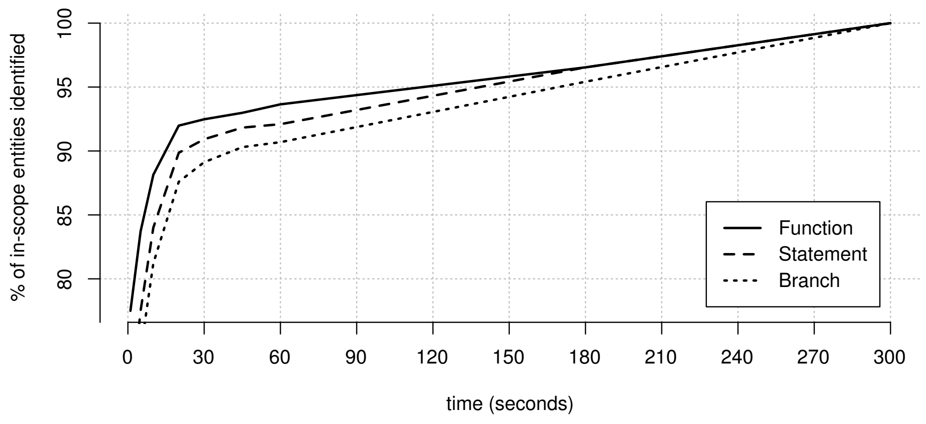
With respect to (1), our approach presupposes that the information regarding the specific reuse scope is available. In other words, we assume that developers know which functionalities are going to be reused and this information could be available informally in their minds or in some formal specification document. Some manual intervention may likely be required to express the reuse scope in a format that can be used by the technology adopted for identifying the in-scope entities and this could incur costs that are hard to quantify in the general case. In our case we adopted DSE and this first step was as simple as expressing the reuse scope constraints into concrete parameters for the tool we adopted.

Regarding (2), the in-scope entities identification can then be done in automated way. As previously stated, in our studies we adopted DSE and we decided to allow the DSE tool to explore the source code of each subject for a maximum of 5 minutes, based on a previous research paper [3] that reported that this time budget was enough to allow KLEE to achieve high coverage exploration even for complex and environmentally-intensive programs.

We considered 5 minutes to be a reasonable amount of time for the purposes of our study, as this task is performed just once for each testing scope and the same results can then be used by all the techniques investigated (prioritization, selection, and minimization). However, if such time budget is not doable for some testing scenario this would not compromise the adoption of our approach as the vast majority of the in-scope entities can be identified after just a few seconds of DSE exploration. As we can see in Figure 8, on average, 90% of the in-scope functions and statements can be identified within 30 seconds. For the branch coverage criterion, 90% of the in-scope entities are identified at around 45 seconds.

Although DSE is largely adopted in research, it is known for limited scalability due to classic challenges such as the path explosion, for example [4]. Thus, for adopting our approach for very large software, if the limited scalability cannot be mitigated, a different technique for identifying the in-scope entities should be considered.

Figure 8: In-scope entities identification over a time period of 5 minutes



We now provide some discussion about the costs related to the individual testing techniques investigated in this work. For prioritization techniques one can consider roughly that $C_{exec} = C_{exec'}$ and $C_{act} = C_{act'}$, as prioritization per se does not involve a reduction of the test suite size, but only a modification of its execution ordering. Then we would have that with any prioritization technique (including scope-aided ones) $C_{POST} > C_{ORIG}$, because the former will have some additional C_{tech} associated. It is important to notice, though, that the real benefit provided by prioritization lies in the fact that faults can be revealed faster — which means that developers can start debugging activities as soon as possible — and the *de facto* metric to quantify the gains is the APFD_C.

As reported in Section 6, overall the scope-aided approach improved the APFD_C for all the subjects investigated, with the biggest improvement being an increase of 40.35% in the average APFD_C for gzip using the total approach. Besides that, when considering different fractions of a given test suite (75%, 50%, and 25%) scope-aided prioritization performed better than the not scope-aided one in the majority of the cases. In particular, for the 25% fraction, our approach defeated the traditional one in all the cases considered, even if to a small extent.

Regarding the cost of selection and minimization, in general we have that these aim at decreasing the overall cost of testing by reducing the size of test suite T'. This is of course true when the savings in terms of decreased $C_{exec'} < C_{exec}$ and $C_{act'} < C_{act}$ are higher than the cost of applying the technique C_{tech}. With our approach we saved, on average, 176 test cases for grep (22 tests in addition to those saved by the traditional approach); 190 test cases for gzip (11 extra test cases); and 355 tests for sed (14 test cases besides those saved by traditional). These results are displayed in percentage format in Table 6. The average figures for minimization (Table 7) were very similar and the maximum number of extra test cases saved were 19, 30 and 55 for gzip, sed and grep, respectively. Again, deciding whether the potential saving is worthwhile against the cost of the technique, will depend on the specific test environment and the cost incurred into executing and analyzing each test case.

10. Threats to Validity

In this section we present a summary of the potential threats to validity of our study, including: threats to internal validity (concern aspects of the study settings that could bias the observed results); threats to external validity (concern aspects of the study that may impact the generalizability of results); and threats to construct validity (concern confounding aspects by which what we observed is not truly due to the supposed cause).

10.1. Threats to internal validity

Testing scenarios representativeness: We defined the testing scopes for this study based on possible realistic uses of the subjects chosen. Real use of the subjects may include different scenarios that had not been considered in this study. We controlled this threat by carefully reading the subjects' documentation to understand well how they could be used before selecting the testing scopes for our study.

In-scope entities identification: As stated, we used KLEE for performing the symbolic execution of the subjects' code and identifying the set of in-scope entities. To deal with the path explosion, a classic challenge for symbolic execution [4], it is usually necessary to define a time budget within which the path exploration will occur. Because some of the KLEE's search heuristics for path-finding are random-based, if the exploration is halted by a timeout, the output is not deterministic (i.e., a different set of in-scope entities could be identified in different runs). To minimize this threat, we allowed KLEE to explore our subjects' code for 5 minutes expecting that the majority of the paths would had been explored within this time budget (in [3], the authors reported that some of the `GNU Coreutils` programs achieved high coverage exploration within three minutes).

SIR test suite coverage: The SIR test suites used for our investigations does not achieve full coverage of the study subjects. Different results could had been achieved if full coverage was provided, as the level of coverage may affect effectiveness, see, e.g., [20]. One possible way of controlling this threat would be adding more test cases to achieve different levels of coverage up to full. We performed a cost-benefit analysis and decided to use each subject as it is provided (i.e., not to introduce other test cases) with all of its artifacts, since it represents the *golden standard* for benchmarking purposes.

In-scope faults identification: The set of in-scope faults derived for computing the APFD_C and for evaluating the impact on the fault detection capability may not contain all the relevant faults that could be possibly revealed in a given testing scope. However, since we use the same fault matrix for both the original approaches and the scope-aided ones, we do not see how such threat could produce different impacts on different evaluations in systematic way, and thus influence the results on fault detection rates.

Prioritization evaluation: In this study we used APFD_C as the metric for evaluating the speed in which faults are revealed by a given test suite. However, APFD_C is not the only possible measure of rate of fault detection. Control for this threat can be achieved only by conducting additional studies using different metrics for evaluating the prioritization quality of the test suites used in our study.

Selection and Minimization evaluation: We adopted two metrics for evaluating the selection and minimization of the test suites in our study: (i) test suite reduction and (ii) impact on fault detection capability. However, there exist other metrics that could be adopted to evaluate the quality of the produced test suites. Control for this threat can be achieved only by conducting additional studies using different metrics for evaluating the selection and minimization quality.

10.2. Threats to external validity

Subject representativeness: In this work we investigated the proposed scope-aided testing approaches on 17 variant versions of C programs. We acknowledge that the 17 versions belong to three subjects only. However it is important to notice that, in some cases, the differences between versions of a same subject were quite significant. Indeed, when considering the case of sed, for example, the development of v5 spans almost 5 years and the differences when compared with v4 are astounding — nearly every "major" function has been changed significantly. So, these could be considered as different programs. A similar case of fairly significant differences happened between versions v6 and v7 (as reported in the accompanying material from the SIR repository). Nevertheless, additional studies using a range of diversified subjects should be conducted for better representativeness. As explained in Section 5.1 the study settings imposed a set of requirements on the subjects that made it not easy to identify good candidates. Besides that, as stated before, our experiments were computationally intensive and very time-consuming.

Faults representativeness: In our study we considered mutant faults and subjects with real faults might yield different results. Control for this threat can be achieved only by conducting additional studies using subjects with real faults.

Prioritization strategies: We assessed scope-aided prioritization when used as a boost for coverage-, similarity-, and search-based prioritization strategies. Other prioritization techniques and more instances of the strategies already studied should be investigated before more general conclusions can be drawn.

Selection and Minimization strategies: Analogously to the previous threat, in this study we investigated only one selection (and one minimization) strategy. Other strategies should be investigated before more general conclusions can be drawn. Control for this threat can be achieved only by conducting further studies using different selection (and minimization) techniques.

10.3. Threats to construct validity

Study design: We introduced scope-aided testing approaches for prioritization, selection, and minimization that focus on giving priority to the detection on the most relevant (in-scope) faults. In the study we wanted to evaluate the fault detection rate (for prioritization) and the impact on fault detection capability (for selection and minimization) of the original and the scope-aided approaches on the targeted subset of in-scope faults. We were conscious that constructing such a study could suffer of biased design, as we were using our testing approaches' goal (the in-scope faults) also into the evaluation metrics (APFD_C and impact on fault detection capability). The only way to prevent this threat would have been to use study subjects with failure reports from real world usage, but we were not able to find such subjects. To mitigate the threat, we used large suites of random test cases, different from the baseline test suite (the one that is later prioritized, selected and minimized), to identify the in-scope faults.

11. Related Work

In this work we have introduced the scope-aided testing approach to support test prioritization, selection and minimization of reused software.

Our work builds on a huge literature of white-box approaches for test prioritization, selection and minimization, as surveyed in [47] and [5]. However, to the best of our knowledge, techniques and tools in the literature are mostly conceived for regression testing purpose, and there exists no previous work that expressly targets test prioritization, selection and minimization when an existing test suite is used to test a software component or a piece of code that is reused in a new context. To confirm such statement, we have made a systematic search within both ACM DL and IEEExplore repositories, looking for any paper that included in the abstract both terms "reuse" and "test" (or "testing"), and any of the three words: prioritization, selection or minimization. We then screened all papers returned by the search, and in fact among them we could not find any relevant reference.

In test suite prioritization test cases can be re-ordered based on various criteria [9, 18]. So far criteria used by other authors include test case execution time, history of failure detection, fault localization costs, and others, but to the best of our knowledge no work exists that relates program entities to their relevance in a reuse context and uses such information to re-order them, as we do. Our approach prioritizes white-box test cases based on possible constraints on the input domain in order to focus on more relevant faults in a reuse scope. As such, scope-aided prioritization is also related to operational testing, as pursued in, e.g., Musa's SRET (Software Reliability-Engineered Testing) approach [36]. In SRET test cases are selected from the user's operational profile, thus those inputs that are forecast to be more often invoked in use are also more stressed in testing. Hence, also SRET could be deemed as a test case prioritization method. In a similar way to SRET, by prioritizing test suites focusing on the in-scope entities, we also aim at targeting the input subdomain that is the most relevant for the user, while giving lower priority to inputs that are not or seldom exercised. Differently from SRET, we do not use a statistical approach to order the test cases, but exploit dynamic analysis to identify the program entities that are covered when the in-scope inputs are invoked.

Achieving adequate coverage may require a high number of test cases, and researchers have proposed several approaches for test selection and minimization, e.g. [25, 14]. Scope-aided approaches for test selection and minimization during reuse testing aim at reducing the size of a test suite based on the principle to retain those test cases covering the code entities that really matter, and discard test cases covering entities that are outside scope.

While software reuse may involve reuse of software artifacts and documentation beyond mere code reuse, our work is related to reuse of code, both in systematic [12] or pragmatic forms [11]. Indeed, depending on the extent and formality under which code is reused, testing difficulties and approaches vary. The research of testing in software reuse has followed two main directions: product-lines testing and component-based testing. As surveyed in [44, 11], various testing frameworks have been proposed for product lines. The main challenge in product line testing is to account for variability among products, and not among different usage contexts or scopes. Therefore, we consider our approach not closely related to product line testing. In component-based testing [15] researchers have investigated both approaches for testing and certifying one component built for reuse, and approaches for testing the integration of a component into a system [21]. Our research is more related to the second case, in that our approach can be used for testing the integration of a component into a system, assuming that the component code is available, as is generally the case in internal reuse. A component developer following the approach proposed in [37], could incorporate in a component metadata relevant information to facilitate the derivation of possible usage scenarios in component reuses. In [48] a framework is presented that supports prioritization of compatibility testing in component integration: the aim is close to our goal, however prioritization focuses on evolution and regression, and not on the usage context.

12. Summary, Conclusions and Future Work

In front of a huge literature on challenges and approaches on the one side to support and promote software reuse, and on the other side to improve software testing cost-effectiveness, the very problem of testing reused code has surprisingly received scant attention. In particular, while coverage measures are widely used in software development, appropriate coverage testing tools for code that is reused are lacking [45]. There exist many regression testing techniques and tools that address the retesting of software after maintenance in order to ascertain that the modifications have not produced undesired effects. As such, these approaches naturally focus on those test cases that exercise the software functionalities or code entities impacted by the changes. When testing a reused code (be it changed or unchanged) into a new context, the main goal is to test the ways in which the software is invoked from the new context. Quoting Weyuker [46], *when developing a component for a particular project or application, . . . testers usually have some information or intuition about how the software will be used and therefore emphasize, at least informally, testing of what they believe to be its central or critical portions. These priorities will likely change, however, if it is decided to incorporate the component into a different software system.*

The approach of scope-aided testing we propose here entails exactly leveraging the information or intuition about how the software will be used in each new context or scope to redirect the focus of existing white-box prioritization, selection and minimization, and thus make them more cost-effective for reuse testing.

In brief our scope-aided testing approach requires developers to make explicit any constraints limiting the reuse input domain. The identification of reuse constraints is the first basic step, and the only one which obviously requires human intervention. After the scope constraints are given, the approach can be completely automated, and in this paper we have leveraged in particular dynamic symbolic execution techniques.

We use such constraints first to identify in-scope entities (as described in Section 4), and then to adapt an existing technique to privilege these in-scope entities when picking the next test case to execute. We have shown how to do so considering some existing prioritization (Section 6), selection (Section 7) and minimization (Section 8) approaches, however the basic idea is general and other techniques could be adapted as well.

The results from our empirical evaluation are encouraging: although scope-aided did not win consistently in all studies, when looking in particular at those cases in which only a limited number of test cases can be executed, scope-aided clearly outperformed not-scope aided approaches. When only 25% of the test cases are executed (see Table 3), for example, scope-aided prioritization defeated, to a small extent, the traditional approaches for all the coverage criteria considered by finding in-scope faults faster. When applied to minimize test suites, scope-aided consistently produced lower impact in terms of reducing the in-scope fault detection capability (see Figure 7). In this paper we have reported summary results from our empirical studies. More detailed data can be found on-line at `http://labsedc.isti.cnr.it/tools/scope-aided-testing`.

Concerning the future, further experimentation is needed to know more about the usefulness of scope-aided approaches to reuse testing. We would like to address other subjects and other test techniques, as well as to study other possible methods to identify in-scope entities. We aim at increasing the data reported in the above mentioned page as more subject are studied.

The approach proposed in this work can only be applied in those reuse contexts in which the source code of the reused software is available. However, the studies conducted here did not take into account the impact of changes in the reused code to the scope-aided approach. Our intuition is that our approach can still be applicable and useful, but we would need to run additional experiments in a setting that explicitly models code changes and assess its cost-effectiveness in order to confirm (or not) this intuition.

Besides, we envisage that the same idea of scope could be applied to black-box reuse considering other types of test requirements, for example the specification of functional requirements. In the same way that we distinguished here between in-scope and out-of-scope code entities, we could identify in-scope requirements and adapt the functional testing of a reused code accordingly. Similarly, one could think of prioritizing or selecting test cases addressing configuration testing, by privileging those configurations that are more relevant in a reuse context.

Acknowledgements

Breno Miranda wishes to thank the Brazilian National Council for Scientific and Technological Development (CNPq) for providing his scholarship grant.

[1] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments?[software testing]. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 402–411. IEEE, 2005.

[2] M. Balcer, W. Hasling, and T. Ostrand. Automatic generation of test scripts from formal test specifications. In *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification*, TAV3, pages 210–218, New York, NY, USA, 1989. ACM.

[3] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[4] C. Cadar and K. Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, Feb. 2013.

[5] C. Catal and D. Mishra. Test case prioritization: A systematic mapping study. *Software Quality Control*, 21(3):445–478, Sept. 2013.

[6] T. Chen and M. Lau. Heuristics towards the optimization of the size of a test suite. In *Proceedings of the 3rd international conference on software quality management*, volume 2, pages 415–424, 1995.

[7] T. Chen, X. song Zhang, S. ze Guo, H. yuan Li, and Y. Wu. State of the art: Dynamic symbolic execution for automated test generation. *Future Gen. Comp. Systems*, 29(7):1758 – 1773, 2013.

[8] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[9] H. Do and G. Rothermel. An empirical study of regression testing techniques incorporating context and lifetime factors and improved cost-benefit models. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, pages 141–151, New York, NY, USA, 2006. ACM.

[10] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 329–338, Washington, DC, USA, 2001. IEEE Computer Society.

[11] E. Engström and P. Runeson. Software product line testing - a systematic mapping study. *Inf. Softw. Technol.*, 53(1):2–13, Jan. 2011.

[12] W. Frakes and S. Isoda. Success factors of systematic reuse. *Software, IEEE*, 11(5):14–19, Sept 1994.

[13] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Trans. Softw. Eng.*, 14(10):1483–1498, Oct. 1988.

[14] G. Fraser and F. Wotawa. Redundancy based test-suite reduction. In M. Dwyer and A. Lopes, editors, *Fundamental Approaches to Sw Eng.*, volume 4422 of *LNCS*, pages 291–305. Springer, 2007.

[15] J. Gao and Y. Wu. Testing component-based software – issues, challenges, and solutions. In R. Kazman and D. Port, editors, *COTS-Based Software Systems*, volume 2959 of *Lecture Notes in Computer Science*, pages 2–2. Springer Berlin Heidelberg, 2004.

[16] V. Garousi and T. Varma. A replicated survey of software testing practices in the canadian province of alberta: What has changed from 2004 to 2009? *Journal of Systems and Software*, 83(11):2251 – 2262, 2010.

[17] D. Hao, L. Zhang, L. Zhang, G. Rothermel, and H. Mei. A unified test case prioritization approach. *ACM Trans. Softw. Eng. Methodol.*, 24(2):10:1–10:31, Dec. 2014.

[18] M. Harman. Making the case for MORTO: Multi objective regression test optimization. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 111–114, March 2011.

[19] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2(3):270–285, July 1993.

[20] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proc. of the 36th International Conference on Software Engineering*, ICSE 2014, New York, NY, USA, 2014. ACM.

[21] M. Jaffar-ur Rehman, F. Jabeen, A. Bertolino, and A. Polini. Testing software components for integration: A survey of issues and techniques: Research articles. *Softw. Test. Verif. Reliab.*, 17(2):95–133, June 2007.

[22] Y. Jia and M. Harman. MILU: A customizable, runtime-optimized higher order mutation testing tool for the full C language. In *Proceedings of the 3rd Testing: Academic and Industrial Conference Practice and Research Techniques (TAIC PART'08)*, pages 94–98, Windsor, UK, 29-31 August 2008.

[23] B. Jiang, Z. Zhang, W. K. Chan, and T. Tse. Adaptive random test case prioritization. In *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on*, pages 233–244. IEEE, 2009.

[24] D. S. Johnson. Approximation algorithms for combinatorial problems. In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, STOC '73, pages 38–49, New York, NY, USA, 1973. ACM.

[25] J. Jones and M. Harrold. Test-suite reduction and prioritization for modified condition/decision coverage. In *Proc. IEEE Int. Conf. on Software Maintenance*, pages 92–101, 2001.

[26] J. Kasurinen, O. Taipale, and K. Smolander. Test case selection and prioritization: Risk-based or design-based? In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, page 10. ACM, 2010.

[27] R. Land, D. Sundmark, F. Lüders, I. Krasteva, and A. Causevic. Reuse with software components - a survey of industrial state of practice. In *Proceedings of the 11th International Conference on Software Reuse: Formal Foundations of Reuse and Domain Engineering*, ICSR '09, pages 150–159, Berlin, Heidelberg, 2009. Springer-Verlag.

[28] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, July 1993.

[29] Z. Li, M. Harman, and R. M. Hierons. Search algorithms for regression test case prioritization. *Software Engineering, IEEE Transactions on*, 33(4):225–237, 2007.

[30] J.-L. Lions. Ariane 5 flight 501 failure, 1996.

[31] M. D. McIlroy. Mass-produced software components. In *Software Engineering: A Report on a Conf. Sponsored by the NATO Science Committee*, pages 138–155. NATO, 1969. `http://www.cs.dartmouth.edu/~doug/components.txt`.

[32] B. Miranda and A. Bertolino. Improving test coverage measurement for reused software. In *Software Engineering and Advanced Applications (SEAA), 2015 41st Euromicro Conference on*, pages 27–34, Aug 2015.

[33] P. Mohagheghi, R. Conradi, O. M. Killi, and H. Schwarz. An empirical study of software reuse vs. defect-density and stability. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 282–292, Washington, DC, USA, 2004. IEEE Computer Society.

[34] M. Morisio, M. Ezran, and C. Tully. Success and failure factors in software reuse. *IEEE Trans. Softw. Eng.*, 28(4):340–357, Apr. 2002.

[35] G. C. Murphy, D. Notkin, W. G. Griswold, and E. S. Lan. An empirical study of static call graph extractors. *ACM Trans. Softw. Eng. Methodol.*, 7(2):158–191, Apr. 1998.

[36] J. D. Musa. Software-reliability-engineered testing. *IEEE Computer*, 29(11):61–68, 1996.

[37] A. Orso, M. Jean, and D. Rosenblum. Component metadata for software engineering tasks. In W. Emmerich and S. Tai, editors, *Engineering Distributed Objects*, volume 1999 of *Lecture Notes in Computer Science*, pages 129–144. Springer Berlin Heidelberg, 2001.

[38] C. S. Pasareanu and W. Visser. A survey of new trends in symbolic execution for software testing and analysis. *Int. J. Softw. Tools Technol. Transf.*, 11(4):339–353, Oct. 2009.

[39] J. Poulin, J. Caruso, and D. Hancock. The business case for software reuse. *IBM Systems Journal*, 32(4):567–594, 1993.

[40] R. Prieto-Diaz. Status report: software reusability. *Software, IEEE*, 10(3):61–66, May 1993.

[41] T. Ravichandran and M. A. Rothenberger. Software reuse strategies and component markets. *Commun. ACM*, 46(8):109–114, Aug. 2003.

[42] G. Rothermel, R. Untch, C. Chu, and M. Harrold. Prioritizing test cases for regression testing. *Software Engineering, IEEE Transactions on*, 27(10):929–948, Oct 2001.

[43] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Test case prioritization: An empirical study. In *Proc. IEEE Int. Conf. on Software Maintenance, 1999.(ICSM'99)*, pages 179–188. IEEE, 1999.

[44] A. Tevanlinna, J. Taina, and R. Kauppinen. Product family testing: A survey. *SIGSOFT Softw. Eng. Notes*, 29(2):12–12, Mar. 2004.

[45] R. Torkar and S. Mankefors. A survey on testing and reuse. In *Software: Science, Technology and Engineering, 2003. SwSTE '03. Proceedings. IEEE International Conference on*, pages 164–173, Nov 2003.

[46] E. Weyuker. Testing component-based software: a cautionary tale. *Software, IEEE*, 15(5):54–59, Sep 1998.

[47] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *Softw. Test. Verif. Reliab.*, 22(2):67–120, Mar. 2012.

[48] I. Yoon, A. Sussman, A. M. Memon, and A. A. Porter. Testing component compatibility in evolving configurations. *Information & Software Technology*, 55(2):445–458, 2013.