SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2016; **26**:402–426
Published online 21 March 2016 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/stvr.1603

Prioritizing test cases for early detection of refactoring faults

Everton L. G. Alves[1,*,†], Patrícia D. L. Machado[1], Tiago Massoni[1] and Miryung Kim[2]

[1] *Federal University of Campina Grande Campina Grande, 58429-900, Brazil*
[2] *University of California Los Angeles, CA, 90095 USA*

SUMMARY

Refactoring edits are error-prone, requiring cost-effective testing. Regression test suites are often used as a safety net for decreasing the chances of behavioural changes. Because of the high costs related to handling massive test suites, prioritization techniques can be applied to reorder test case execution, fostering early fault detection. However, traditional prioritization techniques are not specifically designed for detecting refactoring-related faults. This article proposes refactoring-based approach (RBA), a refactoring-aware strategy for prioritizing regression test cases. RBA reorders an existing test sequence, using a set of proposed refactoring fault models that define the refactoring's impact on program methods.

Refactoring-based approach's evaluation shows that it promotes early detection of refactoring faults and outperforms well-known prioritization techniques in 71% of the cases.

Moreover, it prioritizes fault-revealing test cases close to one another in 73% of the cases, which can be useful for fault localization. Those findings show that RBA can considerably improve prioritization of test cases during perfective evolution, both by increasing fault-detection rates as well as by helping to pinpoint defects introduced by an incorrect refactoring.

Received 10 September 2014; Revised 13 February 2016; Accepted 24 February 2016

KEY WORDS: test case prioritization; refactoring; automated software testing

1. INTRODUCTION

Refactorings aim to improve the internal aspects of a program while preserving its external behaviour [1, 2]. Refactoring edits are very common in software development; Xing and Stroulia [3] report almost 70% of all structural changes in Eclipse's history are refactoring related. Recent studies find software quality gains due to refactoring. Kim *et al*. analysis on Windows 7 versions [4] shows a significant reduction of inter-module dependencies and post-release defects in refactored modules. Moreover, Investigation of MacCormack *et al*. with the evolution of Mozilla's architecture [5] detects a significant improvement after continuous refactoring.

Although popular IDEs include built-in refactoring tools, developers still perform most refactorings manually. Murphy *et al*. [6] find that about 90% of refactoring edits are manually applied. Negara *et al*. [7] show that expert developers prefer manual refactoring over automated. The underuse of refactoring tools is mostly due to usability issues, to the lack of trust, and to developer's unawareness [8, 9]. Moreover, recent studies have shown that even well-known refactoring tools are not free of problematic refactorings [10, 11].

As subtle faults may pass unnoticed, either manual or automatic refactorings require validation. By subtle faults, we mean edits that alter a program's behaviour without generating compilation errors. Dig *et al*. [12] state that nearly 80% of the changes that break client applications are API-level refactoring edits. Studies using version histories find that there is a relationship between the number

*Correspondence to: Everton, L. G. Alves, Federal University of Campina Grande, Campina Grande, 58429-900, Brazil.
†E-mail: everton@copin.ufcg.edu.br

of refactorings and software bugs [13, 14]. In addition, 77% of the participants from the survey of Kim *et al*. with Microsoft developers [4] confirm that refactoring may induce the introduction of subtle bugs and functionality regression.

A number of strategies are designed to prevent behavioural changes when refactoring: (i) *refactoring mechanics*, proposed by Fowler [1], guide the application of refactoring with the combination of micro changes and compilation/test checks; (ii) *the formal specification of refactoring edits*, founded by theories of object-oriented programming [15–17]; (iii) *refactoring engines*, automate the application of refactoring edits by checking pre-conditions (e.g. Eclipse[‡], NetBeans[§], JRRT[¶]); and (iv) *regression testing*, test suites are used to increase confidence on behaviour preservation after refactoring [18].

From those options, regression testing is probably the most popular alternative. However, as a system evolves, its regression test suite tends to increase, because new test cases can be added to check new functionalities [19, 20]. Thus, it may be impractical to rerun regression suites after each refactoring when working with a large test suite—it can take a long time for the first test case to fail as well as it may be difficult and costly to gather enough information on test cases that fail so that fault localization can begin. In such context, there is a need for techniques that preserve test effectiveness, with as few test cases as possible. Test case prioritization [21] rearranges a test suite aiming to improve achievement of certain testing goals (e.g. the rate of fault detection). Several prioritization techniques have been proposed [22–28]; most consider code coverage as prioritization criteria [21]—some are detailed in Section 3, and thoroughly in Section 7.

Although general-purpose (or traditional) solutions might produce acceptable results, specific problems may require particular and/or adaptive solutions [29–33]. In previous empirical studies [34–36], we investigate how traditional general-purpose techniques behave when dealing with seeded refactoring faults in the context of real-open source projects. This investigation shows that those techniques perform poorly when aiming to anticipate the detection of refactoring faults—fault-revealing test cases were placed in the top of the prioritized suite only in 35% of the cases (Section 2 exemplifies such scenarios). Traditional prioritization approaches lack useful information for better scheduling test cases that detect refactoring faults. To the best of our knowledge, there is no prioritization technique specialized in refactoring fault detection. This article proposes the *refactoring-based approach* (RBA), a technique for prioritizing test cases guided by refactoring edits. This technique's prioritization heuristic assumes that a test case is more likely to detect a refactoring problem if it covers the locality of the edits, and/or commonly impacted methods. In order to relate possibly impacted methods to test cases, *refactoring fault models* (RFMs) for five common refactoring types were defined (rename method, move method, pull up field, pull up method and add parameter) [6, 7]. Those models identify the commonly impacted methods after a specific refactoring. The preliminary ideas behind RBA are introduced in a previous workshop paper [34], while this article extends the solution with a complete definition of RFMs and a broader evaluation.

Refactoring-based approach is evaluated by means of two empirical studies: a case study using three real-open source projects (EasyAccept, JMock and XML-Security) with seeded refactoring faults and a controlled experiment using subtle refactoring faults collected from related studies and extensive test suites. In comparison with six well-known prioritization techniques [37, 38], RBA successfully detects refactoring faults earlier than all traditional prioritization techniques in most cases (71%).

Studies in the literature show that prioritization techniques can impact fault localization effectiveness, particularly if a selection of ranked test cases does not provide enough information to pinpoint faults [39–42]. The reason is that fault localization techniques may not produce good results if the suite has mostly passing tests [42].

Thus, RBA is also investigated regarding how spread the fault-revealing test cases are. By spread, we mean how distant from one another, in the prioritized sequence, the test cases that reveal faults are. The evaluation shows that RBA provides orders with more narrowly spread test cases in 73%

[‡]http://eclipse.org/
[§]http://netbeans.org/
[¶]https://code.google.com/p/jrrt/

DOI: 10.1002/stvr

of the cases. This new arrangement may provide developers and/or testers earlier and more precise information to understand and locate the fault.

In summary, the main contributions of this article are as follows:

- A test case prioritization technique centred on refactoring edits (RBA—Section 4), based on a number of RFMs—Section 5) for some of the most common refactoring edits in Java programs. RBA is integrated to an open source test case prioritization tool, PriorJ[‖];
- An evaluation of the technique by means of two empirical studies (Section 6), a case study with three open source projects and seeded subtle refactoring faults and a controlled experiment with subtle refactoring problems and extensive test suites. The studies provide statistical evidence that RBA fosters early detection of refactoring-related behavioural changes.
- A new metric for evaluating how spread is the fault-revealing test cases throughout a prioritized test sequence. Besides speeding up the rate of fault detection, a good prioritization technique would also place all fault revealing test cases in early and close positions. This new metric (*F-spreading*) measures this important aspect about prioritization.

Despite the fact that RBA is highly motivated by the scope of manual refactoring, it can also be applicable with automated refactoring. For instance, refactoring tools are not bug free [10, 11], and developers often wish to double check behaviour preservation after automated refactoring. Furthermore, RBA can add value to the validation process by prioritizing and discarding obsolete test cases. These tasks are very hard to do manually, as developers often give up refactoring if they cannot manage their test suite. Moreover, several approaches use extensive test suites for automatically validating refactorings [18, 43]. RBA could be used for reorganizing and/or guiding such automatically generated test suites.

2. MOTIVATIONAL STUDIES

This section presents two motivational studies that illustrate the limitations of state-of-art prioritization techniques in promoting early detection of behavioural changes after refactoring edits. Consider the JMock open source project[**], a library that supports test-driven development of Java code with mock objects. The source is about 5 KLOC, with a test suite of 504 JUnit test cases that cover 92.1% of the statements. Suppose John participates in JMock and decides to perform a pull up field refactoring edit. In a disciplined manner, John decides to follow the pull up field mechanics, as defined by Fowler [1]:

1. Inspect all uses of the candidate fields to ensure they are used in the same way.
2. If these fields have different names, rename them for establishing a uniform name for the superclass field.
3. Compile and test.
4. Create a new field in the superclass.
5. Delete the subclass fields.
6. Compile and test.

In a busy day, John, by mistake, neglects the first step of Fowler's mechanics and ends up introducing a behavioural change. Instead of moving fields used in the same manner, he moves fields with the same name but with different purposes (Figure 1—code insertion is marked with '+'). Although the two moved fields are named equally—`myActualItems`, in classes `ExpectationSet` and `ExpectationList` (Figure 1(a)) —they are associated with different types, `HashSet` and `ArrayList`. Thus, after pulling up `myActualItems`, despite the absence of compilation errors, a subtle behavioural change is introduced. This change leads four test cases from JMock test suite to fail.

By rerunning JMock's original test suite, John sees a failure after 400 test cases are run, which can be overwhelming depending on the availability of resources (e.g. test cases with massive database

[‖]https://sites.google.com/a/computacao.ufcg.edu.br/priorj/
[**]http://jmock.org/

DOI: 10.1002/stvr

access tend to be time-consuming and costly). Moreover, considering the suite will be rerun a few times over before the fault gets fixed, significant time will be spent examining failing test cases to find the fault, which imposes a challenge for validating the edit.

In this scenario, consider the following small study. Aiming at anticipating fault detection, JMock test suite is prioritized according to five traditional prioritization techniques: four coverage-based [21] —*total statement coverage, total method coverage, additional statement coverage* and *additional method coverage*—and *random choice*. Table I shows the position of the first test case that reveals John's fault (F-measure), and the average percentage faults detected (APFD) results after prioritization. A higher APFD indicates that a suite is able to detect the faults earlier (more details on these metrics are discussed in Section 6). Although all dispositions improve the rate of fault defection, the best result was produced by random choice.

This result indicates that the heuristics applied by traditional strategies might not be effective to place test cases that reveal refactoring problems in the top positions. By observing failing test cases (one of them is exemplified in Figure 2), one can see they do not access the modified field directly, but the failure takes place due to method calls that are indirectly impacted by the edit. Hence, it is ineffective to use test coverage for detecting this fault, also it can be hard to infer it manually.

Now, suppose that Ann works in the same project as John and, during code maintenance, she decides to decompose a conditional statement [1] within method `org.jmock.Mock.lookupID`. The boolean expression is not readable enough, and she wants to clean it up without changing its meaning. Fowler's steps for this edit are quite simple:

```
public class
    AbstractExpectationCollection{
    ...
}
public class ExpectationSet extends
    AbstractExpectationCollection{
    private HashSet myActualItems =
        new HashSet();
    ...
}
public class ExpectationList extends
    AbstractExpectationCollection{
    protected ArrayList myActualItems
        = new ArrayList();
    ...
}
```

(a) Original code.

```
public class
    AbstractExpectationCollection{
+    private HashSet myActualItems =
    new HashSet();
    ...
}
public class ExpectationSet extends
    AbstractExpectationCollection{
    ...
}
public class extends
    AbstractExpectationCollection{
    ...
}
```

(b) Code after Pull Up Field. Since the field had different types in the subclasses, a behavioral change is introduced after the edit.

Figure 1. An example of a problematic refactoring edit using JMock's code.

Table I. APFD results for Pull Up Field faulty edit.

	F-Measure	APFD
Original Suite	400	0.207
Total Statement Coverage	206	0.592
Total Method Coverage	259	0.487
Additional Statement Coverage	159	0.685
Additional Method Coverage	112	0.778
Random Choice	70	0.862

```
public void testManyFromIterator(){
    Vector expectedItems = new Vector();
    expectedItems.addElement("A");
    expectedItems.addElement("B");
    Vector actualItems = (Vector)expectedItems.
        clone();

    myExpectation.addExpectedMany(expectedItems.
        iterator());
    myExpectation.addActualMany(actualItems.
        iterator());
    myExpectation.verify();
}
```

Figure 2. Failing test case due to the pull up field edit.

DOI: 10.1002/stvr

1. Extract the condition into its own method.
2. Extract the *then* part and the *else* part into their own methods.

Suppose that during this change, when extracting the conditional to the new method, Ann, by mistake, ends up changing its meaning—instead of verifying the existence of `id`, the condition now only checks whether it is null (Figure 3). As no compilation error occurs, Ann does not notice that a behavioural change has been introduced.

By running JMock's original test suite, the single test case revealing this problem is in position 335. Even after prioritization, the results from the traditional coverage-based techniques do not improve this scenario significantly (Table II).

Looking closely at the failing test case (Figure 4), one can observe it does not directly call (`lookupID`). Hence, fault localization is indeed harder. Prioritization by traditional coverage-based techniques can be ineffective, as they do not consider anything but coverage data, yielding sequential arrangements in which adjacent test cases have little semantic relationship.

These motivational studies indicate that traditional prioritization techniques often fail to anticipate the detection of refactoring-related faults. A few technical reports [34–36] are provided with a more detailed discussion on those studies.

```
1 public MatchBuilder lookupID(String
      id){
2     if (!idTable.containsKey(id)){
3         throw new
              AssertionFailedError("no
              expected invocation named
               '"+ id +"'");
4     }
5     return (MatchBuilder) idTable.get
          (id);
6 }
```

(a) Original code.

```
1 public MatchBuilder lookupID(String
      id){
2     if (newMethod(id)){
3         throw new
              AssertionFailedError("no
              expected invocation named
               '"+ id +"'");
4     }
5     return (MatchBuilder) idTable.get
          (id);
6 }
7 +public boolean newMethod(String id){
8 +    return id == null;
9 +}
```

(b) Code after Ann's decompose conditional refactoring. The IF conditional was modified. Because there is no compilation error, Ann did not notice it.

Figure 3. An example of a problematic decompose conditional refactoring edit using JMock's code.

Table II. APFD results Decompose Conditional faulty edit.

	F-Measure	APFD
Original Suite	335	0.336
Total Statement Coverage	61	0.879
Total Method Coverage	61	0.879
Additional Statement Coverage	93	0.816
Additional Method Coverage	307	0.391
Random Choice	396	0.215

```
1  public void testDetectsMissingIDs(){
2      String missingID = "MISSING-ID";
3      try{
4          mock.stubs().method("hello").after(
               missingID);
5      } catch (AssertionFailedError ex){
6          AssertMo.assertIncludes("error message
               contains missing id", missingID, ex.
               getMessage());
7          return;
8      }
9      fail("should have failed");
10 }
```

Figure 4. Failed test case due to the problematic decompose conditional edit.

DOI: 10.1002/stvr

3. TEST CASE PRIORITIZATION

Rothermel *et al*. [21] formally define the prioritization problem as follows:
Given: T, a test suite; PT, the set of permutations of T, and f, a function from PT to real numbers.
Problem: Find $T' \in PT$ such as $\forall T'' \in PT \bullet T'' \neq T' \rightarrow (f(T') \geqslant f(T''))$, where PT represents the set of possible orderings of T, and f is a function that calculates the best results when applied to any ordering.

Test case prioritization aims at proposing a specific execution order for achieving some testing goal. This goal—formalized by function f in the definition earlier—varies according to each domain (e.g. increasing the rate of fault detection or accelerating achievement of a coverage ratio). In practice, however, depending on the testing goal, the test case prioritization problem may be intractable [37]. Hence, prioritization solutions are usually based on heuristics.

Because of their simplicity, and their satisfactory results in general, coverage-based prioritization techniques, and their variations, are the most used in practice [44]. Coverage-based techniques are based on the idea that, the higher is a test's coverage, the more likely it is to reveal faults. Some of the most used techniques are briefly described as follows— they are used as baseline in the empirical studies presented in this paper (Section 6).

- *Total statement coverage* (TSC): schedules test cases according to their statement coverage;
- *Total method coverage* (TMC): similar to TSC, but considering method coverage;
- *Additional statement coverage* (ASC): first selects the test case with the highest statement coverage; second, the test case with the highest coverage of *statements not covered by the previously selected test cases* is chosen next. This process is repeated until the test suite is completely reordered;
- *Additional method coverage* (AMC): similar to ASC, but considering method coverage;
- *Random* (RD): produces a randomly-ordered suite;
- *Change blocks* (CB) [38]: identifies changed statements between two versions of a program and schedules test cases according to their coverage of those statements.

4. THE REFACTORING-BASED APPROACH

The RBA is a prioritization technique for early uncovering of refactoring-related behavioural changes by a regression test suite.

Figure 5 provides an overview of RBA, in which round-edged rectangles represent activities, dotted rectangles are input or output artefacts, and arrows indicate a flow between activities or a relationship between activities and artefacts.

Inputs and outputs RBA requires inputs usually available when a refactoring task is performed:

- *Original and refactored versions of a program*. The *base* version—a stable version of the program for which all tests from the regression suite have passed—and the *delta* version—the version after refactoring edit(s).
- *A test suite*. A set of test cases that reflects the behaviour of the *base* version.

As output, RBA generates *a prioritized test suite*: the same test cases from the original suite but in a new execution order.

Guiding example In the following sections, RBA is presented along with a guiding example. Suppose a developer performs the pull up method edit shown in Figure 6. Method `k(int i)` is moved from class `B` to its superclass `A`. Yet, simple, this change introduces a behavioural change; method `B.m` produces a different output (`10` in Figure 6(a) and `20` in Figure 6(b)).

Approach Each refactoring imposes a different set of changes, and several types of behavioural changes may be introduced. Thus, RBA must work in a different manner depending on the refactoring applied.

DOI: 10.1002/stvr

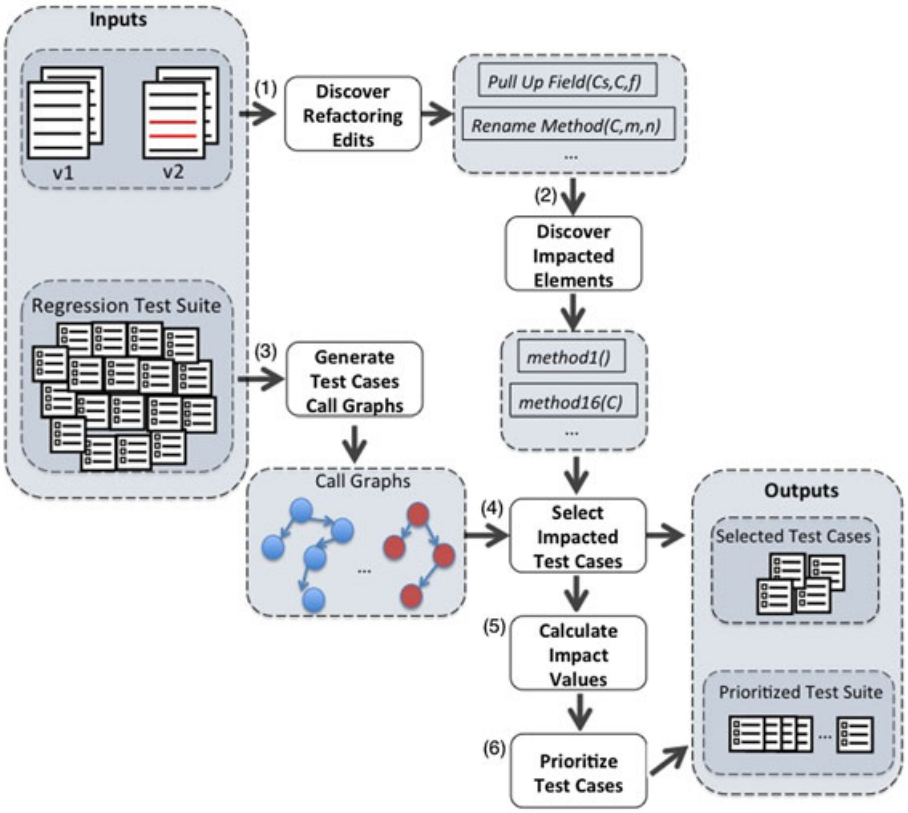

Figure 5. Refactoring-based approach overview.

```java
public class A{
  public int k(long i)
  {
   return 10;
  }
  public int x(){
   return 5;
  }
  public int sum(){
   int xRes = x();
   return xRes +
     k(xRes);
  }
}
public class B
     extends A{
  public int k(int i){
   return 20;
  }
  public int m(){
   return
     new A().k(2);
  }
  public int a(){
   return 30;
  }
}
```

(a) Original code.

```java
public class A{
  public int k(long i)
       {
   return 10;
  }
+ public int k(int i)
+  {
+   return 20;
+  }
  public int x(){
   return 5;
  }
  public int sum(){
   int xRes = x();
   return xRes
     + k(xRes);
  }
}
public class B
     extends A{
  public int m(){
   return
     new A().k(2);
  }
  public int a(){
   return 30;
  }
}
```

(b) Code after pulling up *k(int)*. Now method overloading in class A makes *B.m* run differently.

```java
public void test1(){
  p1.B b = new p1.B();
  int res = b.m();
  assertEquals (10,
       res);
}
public void test2(){
  p1.B b = new p1.B();
  int res = b.a();
  assertEquals (30,
       res);
}
public void test3(){
  p1.B b = new p1.B();
  int res = b.sum();
  assertEquals (15,
       res);
}
```

(c) JUnit test cases. *test1* fails in its assertion.

Figure 6. An example of a problematic refactoring edit.

DOI: 10.1002/stvr

Table III. Ref-strings for five refactoring types.

Refactoring Type	ref-string	Description
Rename Method	*RenameMethod (C, m, n)*	- *C*, name of the class under refactoring; - *m*, former name of the method under refactoring; - *n*, new name of the method under refactoring.
Move Method	*MoveMethod (C1, C2, m)*	- *m*, name of the moved method; - *C1*, name of the original class where *m* was localized; - *C2*, name of the class where *m* is now localized.
Pull up Field	*PullUpField (Cs, C, f)*	- *f*, name of the moved field; - *C*, name of the original class where *f* was localized; - *Cs*, name of a superclass of C where *f* is now localized.
Pull up Method	*PullUpMethod (Cs, C, m)*	- *m*, name of the moved method; - *C*, name of the original class where *m* was localized; - *Cs*, name of a superclass of C where *m* is now localized.
Add Parameter	*AddParameter (C, m, p)*	- *C*, name of the class under refactoring; - *m*, former name of the method under refactoring; - *p*, name of the new parameter.

In activity discover refactoring edits (Figure 5), the applied refactoring is discovered by distinguishing versions *base* and *delta*. A number of techniques can be applied for identifying refactorings (e.g. from a refactoring plan, manually, or with a pair review comparison [45]); RBA reuses a state-of-art refactoring detection tool, Ref-Finder [46].

Ref-Finder identifies, from two consecutive versions of a Java program, which refactoring edits were applied, by means of a technique called template-based refactoring reconstruction. Ref-Finder's output is parsed into *ref-strings*—simplified string patterns that describe type and location of refactoring edits. Their representation resembles a procedure signature; the procedure name represents the refactoring, and parameters are the classes, methods and fields directly involved. Ref-strings for the five refactoring types currently supported in RBA are described in Table III—for instance, `PullUpMethod(B, A, k(int i))` in the guiding example.

With a set of ref-strings, RBA, in activity discover impacted elements, identifies the methods that might have been affected by incorrect refactoring. Because the quality of prioritization is highly related to the accuracy of this identification, RFMs—one for each refactoring—were designed and implemented. They establish the set of methods, collected by static analysis, whose calls are likely to expose undesired behavioural changes, if they exist. The RFM concept is detailed in Section 5.

For each collected *ref-string*, the correspondent RFM algorithm is run, and a set of methods—the affected set (AS)—is built. In the end, AS contains the list of methods from the base version whose behaviour is potentially modified. Back to Figure 6, the application of the pull up method RFM results in `AS = {A.k(long i), A.sum, B.k(int i), B.m}`.

In activity generate test case call graphs, a *dynamic call graph* [47] is generated for each test case in the suite, recovering its hierarchy of method calls.

Each node in the graph represents a method, and each edge `(f,g)` indicates a call from method `f` to method `g`.

Figure 7 shows the call graphs for the test cases in Figure 6(c).

Next, in activity select refactoring-impacted test cases, the graphs are processed for sorting out the most relevant test cases, using as criterion the AS set. The selected test cases are those whose call graphs contain at least one node matching elements from AS. For example, methods `A.k(int i)`, `B.m` and `B.sum` are in both AS and the call graphs of `test1` and `test3`; thus, these test cases get selected.

It is not RBA's original goal, but if, due to project constraints, running a complete test suite is impracticable, test cases absent from this resulting set could be removed (a case of test

DOI: 10.1002/stvr

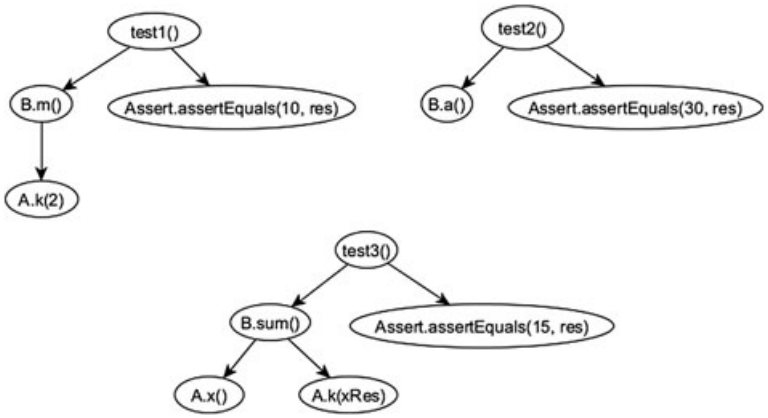

Figure 7. Call graphs of the test cases (Figure 6(c)) for the code under refactoring.

case selection). Yet, depending on how many refactorings are applied, this set may be vast anyway. In this case, prioritization would be relevant as well.

Refactoring-based approach 's prioritization is based on the impact value (IVAL) assigned to each test case during activity calculate impact values. The *IVAL* for each test case is calculated from its call graph; it is the number of methods in AS covered by the test case. For instance, both `test1` and `test3` include two nodes related to elements from the affected set, so they present equivalent impact values ($IVAL_{test1} = 2$, and $IVAL_{test3} = 2$). On the other hand, as no node in *test2* is in AS, this test case has zero impact ($IVAL_{test2} = 0$).

Activity prioritize test cases reorders the suite based on *IVALs*. RBA assumes that test cases covering more elements from AS are more likely to reveal behavioural changes. The test case with the highest *IVAL* is placed on the top of the prioritized test suite then removed from the comparison. Then, the test case with the second highest *IVAL* goes to the second position in the suite, and so on. Ties are dealt with by random choice. Two possible prioritized suites from the example would be as follows: *{test1, test3, test2}* or *{test3, test1, test2}*. For either choice, the behavioural change would be detected by the first test case.

5. REFACTORING FAULT MODEL

An RFM establishes which methods from a program are potentially affected by a specific refactoring edit.

Then, if a refactoring fault is introduced, causing behavioural change, calls to those methods have a significant chance of exposing the fault, given the failure of regression test cases. For each RFM, there is an algorithm to collect those methods. The novelty of RFMs is that individual consequences of applied refactorings are used to improve test case prioritization.

Refactoring fault model definitions catch common faults, such as behavioural changes that usually pass unnoticed even by well-trained developers (e.g. unnoticed overwritten and/or overloaded methods). An initial set of RFMs is defined by combining guidelines for applying refactorings edits in practice (e.g. Fowler's mechanics [1]) with initiatives for defining formal preconditions for sound refactorings (e.g. [15–17]).

Currently, there are RFMs for five of the most common refactoring edits in Java programs [6]—*rename method, move method, pull up field, pull up method* and *add parameter*. For brevity, the following sections present the RFM for two representative refactorings types—*rename method* and *pull up method*—along with their algorithms that build the set of affected methods (AS). The RFMs are illustrated with examples of subtle refactoring problems reported by Soares *et al*. [48]. The remaining three RFMs, and the formalization of all five RFMs by using metamodeling and OCL-based rules, are available at RBA's website [49]. The following auxiliary functions are used to aid the specification of the algorithms.

DOI: 10.1002/stvr

- `searchMethodByName(C,n)`: the method named `n` from class `C`.
- `getCallers(C,m)`: all callers of `m` in class `C`.
- `getSubClasses(C)`: all subclasses of class `C`.
- `getSuperClasses(C)`: all superclasses of class `C`.
- `isStatic(m)`: true if method `m` is static, false otherwise.

5.1. *Rename method*

The RFM for *rename method* is identified by signature `RenameMethod(C, oldName, newName)`, where `C` is the class whose method is to be renamed, `oldName` is the signature of the method to be renamed, and `newName` is the renamed signature.

This RFM specifies the *AS* as including

- `oldName` from `01`;
- all methods in `C ∪ subtypes(C) ∪ supertypes(C)` whose bodies contain at least one call to `oldName` or `newName`. Expression `subtypes(C)` (and `supertypes(C)`, analogously) yields all subclasses that inherit from `C` directly or indirectly;
- if the method `oldName` in `C` is static, all methods, in any class, whose bodies contain a call to `oldName`.

Algorithm 1 describes how to build the *AS* set according to the RFM. In the top Line (1), AS is initialized. Lines (2) to (4) add the method under refactoring and its callers to the AS set. Line (5) adds to AS any method that calls a method with the same signature as the method after refactoring. Lines from (6) to (10) find all subclasses of the refactored class, adding to AS all methods from those classes that call methods with the same signature as *oldName* or *newName*. Lines (11) to (15) do the same for all superclasses of the refactored class. Finally, in Lines (16) to (19), if the original method is static, all classes are examined, adding to AS all methods with direct calls to the static method.

Consider the refactoring depicted in Figure 8—method `B.n` is renamed to `B.k`, generating the target version depicted in Figure 8(b). In this scenario, a behavioural change occurs; method `B.m` from `B` returns 0, instead of 1 as in the previous version. This change is due to a subtle method overriding that could easily pass unnoticed in a large, complex program. To detect such fault, a test case must contain calls to `B.m`, which may invoke `B.k` indirectly, then exercising the distinct behaviour. Line (5) of the RFM's algorithm selects this method as possibly impacted. Also, Lines (4), (8), (13) and (18) identify possibly obsolete test cases calling `B.n`, which is absent in the target version.

```
1  package p1;
2  public class A{
3    public long k(long a){
4      return 1;
5    }
6  }
7  package p2;
8  import p1.*;
9  public class B extends A{
10   protected long n(int a){
11     return 0;
12   }
13   public long m(){
14     return k(2);
15   }
16 }
```

(a) Original code.

```
1  package p1;
2  public class A{
3    public long k(long a){
4      return 1;
5    }
6  }
7  package p2;
8  import p1.*;
9  public class B extends A{
10   protected long k(int a){
11     return 0;
12   }
13   public long m(){
14     return k(2);
15   }
16 }
```

(b) Code after method renaming.

Figure 8. Example of a problematic rename method.

DOI: 10.1002/stvr

Algorithm 1 Rename Method RFM

Require: C class under refactoring;
$oldName$ method to be refactored;
$newName$ refactored method;
1: $AS \leftarrow \emptyset$
2: $m \leftarrow searchMethodByName(C, oldName)$
3: $AS \leftarrow AS \cup \{m\}$
4: $AS \leftarrow AS \cup getCallers(C, oldName)$
5: $AS \leftarrow AS \cup getCallers(C, newName)$
6: $Sub \leftarrow getSubClasses(C)$
7: **for** each class $S \in Sub$ **do**
8: $AS \leftarrow AS \cup getCallers(S, oldName)$
9: $AS \leftarrow AS \cup getCallers(S, newName)$
10: **end for**
11: $Sup \leftarrow getSuperClasses(C)$
12: **for** each class $Sp \in Sup$ **do**
13: $AS \leftarrow AS \cup getCallers(Sp, oldName)$
14: $AS \leftarrow AS \cup getCallers(Sp, newName)$
15: **end for**
16: **if** $isStatic(m)$ **then**
17: **for** each class $Cl \in$ Program **do**
18: $AS \leftarrow AS \cup getCallers(Cl, oldName)$
19: **end for**
20: **end if**
21: **return** AS

5.2. Pull up method

As an RFM, the *pull up method* refactoring is represented by `PullUpMethod(Cs, C,-mName)`, where `Cs` is the class to which `mName` is being moved, and `C` is the original class. The RFM determines the AS set as the following items:

- any method with the same signature as `mName` in `Cs ∪ subtypes(Cs)` (where `C ∈ subtypes(Cs)`);
- all methods in `Cs ∪ subtypes(Cs)` whose bodies contain at least one call to `mName`;
- if `mName` is static, all methods whose bodies contain a call to `mName`.

Algorithm 2, after initializing the *AS* set at Line (1), adds to AS the method to be pulled up plus any other method with the same signature, along with their callers—Lines (2) to (7). Lines from (8) to (12) find the subclasses of the refactored class and add to AS all methods from those classes that call the original method or methods with the same signature. Finally, in Lines (13) to (17), if the original method is static, the algorithm goes through all classes adding to AS every method with direct calls to the moved method.

In Figure 9(b), method `B.test` has its invocation resolved to `B.k`, differently from the source version, in which the call to `k` is resolved to `A.k` The RFM selects, among others, the `B.test` method as possibly affected.

5.3. Limitations of refactoring fault models

The aim of RBA is to anticipate detection of faults when using regression testing; we assume that such approach emphasizes practical constraints over completeness of refactoring fault detection. Therefore, the RFM rules and correspondent algorithms do not intend to perform a complete change impact analysis. More elaborate static analysis, or even fusing some dynamic analysis, could improve analysis (making AS maximal, for instance), but the cost-benefit ratio is yet to be assessed.

DOI: 10.1002/stvr

Algorithm 2 Pull Up Method RFM

Require: C class where the method is originally placed;
Cs class where the method will be moved to;
$mName$ refactored method;

1: $AS \leftarrow \emptyset$
2: $m \leftarrow searchMethodByName(C, mName)$
3: $AS \leftarrow AS \cup \{m\}$
4: $m2 \leftarrow searchMethodByName(Cs, mName)$
5: $AS \leftarrow AS \cup \{m2\}$
6: $AS \leftarrow AS \cup getCallers(C, mName)$
7: $AS \leftarrow AS \cup getCallers(Cs, mName)$
8: $Sub \leftarrow getSubClasses(C)$
9: **for** each class $S \in Sub$ **do**
10: $AS \leftarrow AS \cup getCallers(S, mName)$
11: $AS \leftarrow AS \cup searchMethodByName(S, mName)$
12: **end for**
13: **if** $isStatic(m)$ **then**
14: **for** each class $Cl \in$ Program **do**
15: $AS \leftarrow AS \cup getCallers(Cl, mName)$
16: **end for**
17: **end if**
18: **return** AS

```
1  public class A{
2    public int k(){
3      return 10;
4    }
5  }
6  public class B extends A{
7    public int test(){
8      return k();
9    }
10 }
11 public class C extends B{
12   public int k(){
13     return 20;
14   }
15 }
```

(a) Original code.

```
1  public class A{
2    public int k(){
3      return 10;
4    }
5  }
6  public class B extends A{
7    public int test(){
8      return k();
9    }
10 +  public int k(){
11 +    return 20;
12 +  }
13 }
14 public class C extends B{
15 }
```

(b) Code after a problematic Pull Up method.

Figure 9. Example of a problematic method being pulled up.

Refactoring fault models do not directly model faults but enumerate methods related to a particular refactoring edit. When these methods are exercise by a test case, they might expose a fault. Likewise, RFMs cannot guarantee detection of all types of behavioural changes for a refactoring; we do not expect RFMs to cover every potential defect in such context. RFMs are proposed as heuristics, based on established—theory and practice—literature on refactoring. As such, the approach covers a considerable ground for common (and subtle) refactoring faults. What contributes to this decision is the complexity of anticipating any possible refactoring-related behavioural change within a general-purpose object-oriented language like Java. In fact, theoretical research on defining such completeness property always consider a confined core language [15, 16, 50]. A more comprehensive RFM might lead to excessively large sets of affected methods, possibly, in consequence, decreasing the quality of test case prioritization.

6. EVALUATION

In order to investigate the effectiveness of RBA in the early detection of behavioural changes, two empirical studies were performed: an exploratory case study, in which subtle refactoring faults were

DOI: 10.1002/stvr

seeded into real Java open source projects; and a controlled experiment in which subtle behavioural changes, collected from related research studies, was combined with automatically generated test suites. In both studies, metrics related to how the approach places fault-revealing test cases on suite's top positions were used. Additionally, the second study focused on investigating whether RBA prioritization is capable of placing failing test cases in closer positions.

6.1. Exploratory case study

This case study compared RBA with other prioritization techniques in the context of real Java projects. The goal of this study was to observe RBA's effectiveness when dealing with behavioural changes for a given refactoring edit, from the point of view of a tester.

First, three open source Java projects were selected as experimental objects: XML-Security[††] ($\approx$ 17 KLOC), a library that provides security APIs for manipulating XML documents, such as authorization and encryption; JMock ($\approx$ 5 KLOC), described and used in Section 2; and EasyAccept[‡‡] ($\approx$ 2 KLOC), a tool that helps create and run acceptance tests. These three projects present a common property: they include an active, up-to-date test suite, which is key to this investigation. While JMock's suite comprises 504 JUnit test cases with 92% of code coverage, the two suites used from EasyAccept have 65 and 74 test cases (88% and 87% of coverage, respectively), and XML-Security includes 89 test cases, with 34% code coverage. To provide an overview of the test suites quality, a mutation analysis was performed. For that, the PIT tool [51] was used in its default configuration. This analysis reported the following mutant rates: XML-Security—24%, JMock—82% and EasyAccept—70%. For each experimental object, five faulty versions were created, each version with a single and a distinct seeded subtle refactoring-related behavioural change. For seeding faults, a process, similar to the one described in Section 2, was followed, in which a single step from Fowler's mechanics [1] was neglected. None of the edits introduced compilation errors. Finally, prioritized versions of the suites were created according to seven prioritization techniques from different categories: (i) RBA; (ii) four coverage-based approaches [21] (TSC, TMC, ASC and AMC); (iii) the random approach (RD); and (iv) a modification-based approach [38] (change blocks—CB).

To guide this investigation, the following research question was established: *Can RBA promote early detection of refactoring-related faults?* To address this question, two well-known metrics were used for evaluating prioritized suites: *F-measure* [52, 53], indicating the number of distinct test cases needed to be run for causing the first failure[§§]; and *APFD* [21], measuring the effectiveness of a suite's ordering (Equation 1, where n is the number of test cases, m is the number of exposed faults. TF_i is the position of the first test case which reveals fault i in the ordered test cases sequence). Higher APFD values imply faster fault detection rates.

$$APFD = 1 - \frac{TF_1 + TF_2 + \ldots + TF_m}{nm} + \frac{1}{2n} \tag{1}$$

Tables IV and V summarize the results. In 53% of the cases, RBA enabled fault detection after running a single test case ($F-measure = 1$), which is the best scenario for a prioritized suite. This rate was at least 2.6 times higher than all the other techniques ($CB = 20\%$, $RD = 0\%$, $TSC = 13\%$, $TMC = 13\%$, $ASC = 13\%$, $AMC = 13\%$). Moreover, RBA placed at least one fault-revealing test case among the six first test cases in 93% of the cases, a satisfactory position, given the size of the suites—ranging from 65 to 504 tests. Another result was the high stability of RBA. With exception of one case, APFD for RBA varied over a very tight range [0.925; 0.999], that is, RBA's orderings detected all behavioural changes early, and in similar positions. This conclusion is even more evident when observing the standard deviation of APFD values, leaving out the worst case of

[††] http://xml.apache.org/security

[‡‡] http://easyaccept.sourceforge.net/

[§§] We are using the F-measure metric as introduced by Chen et al. [52] and commonly applied in the test case prioritization community, which is different from the F-score metric, also named F-measure, applied in statistical analysis.

DOI: 10.1002/stvr

Table IV. F-Measure results from the case study.

EasyAccept

Version	F-Measure						
	RBA	CB	RD	TSC	TMC	ASC	AMC
Rename Method	4	41	5	22	19	22	19
Move Method	2	5	23	56	56	56	56
Add Parameter	1	33	12	3	3	3	3
Pull Up Field	39	22	51	35	39	35	39
Pull Up Method	6	6	8	42	37	42	36

JMock

Version	F-Measure						
	RBA	CB	RD	TSC	TMC	ASC	AMC
Rename Method	5	1	105	481	461	467	461
Move Method	1	39	77	28	42	5	15
Add Parameter	1	81	171	481	461	467	461
Pull Up Field	6	41	63	3	2	61	55
Pull Up Method	1	15	140	48	42	9	8

XML-Security

Version	F-Measure						
	RBA	CB	RD	TSC	TMC	ASC	AMC
Rename Method	3	1	19	17	17	17	17
Move Method	1	42	46	1	1	1	1
Add Parameter	1	1	13	1	1	1	1
Pull Up Field	1	29	18	37	27	38	27
Pull Up Method	1	36	16	2	2	2	2

Table V. APFD results from the case study.

EasyAccept

Version	F-Measure						
	RBA	CB	RD	TSC	TMC	ASC	AMC
Rename Method	0.946	0.376	0.930	0.669	0.715	0.669	0.715
Move Method	0.976	0.930	0.653	0.146	0.146	0.146	0.146
Add Parameter	0.992	0.500	0.823	0.961	0.961	0.961	0.961
Pull Up Field	0.479	0.709	0.317	0.533	0.479	0.533	0.479
Pull Up Method	0.925	0.925	0.898	0.439	0.506	0.439	0.520

JMock

Version	F-Measure						
	RBA	CB	RD	TSC	TMC	ASC	AMC
Rename Method	0.99	0.999	0.792	0.046	0.086	0.074	0.086
Move Method	0.999	0.923	0.848	0.945	0.917	0.991	0.971
Add Parameter	0.999	0.840	0.661	0.046	0.086	0.074	0.086
Pull Up Field	0.989	0.919	0.875	0.995	0.997	0.879	0.891
Pull Up Method	0.999	0.971	0.723	0.905	0.917	0.983	0.985

XML-Security

Version	F-Measure						
	RBA	CB	RD	TSC	TMC	ASC	AMC
Rename Method	0.971	0.994	0.792	0.814	0.814	0.814	0.814
Move Method	0.994	0.533	0.488	0.994	0.994	0.994	0.994
Add Parameter	0.994	0.994	0.859	0.994	0.994	0.994	0.994
Pull Up Field	0.994	0.679	0.803	0.589	0.702	0.578	0.702
Pull Up Method	0.994	0.601	0.825	0.983	0.983	0.983	0.983

each technique, ($s_{TSC} = 0.338$; $s_{TMC} = 0.322$; $s_{ASC} = 0.320$; $s_{AMC} = 0.322$; $s_{CB} = 0.180$; $s_{RD} = 0.117$). RBA's standard deviation ($s_{RBA} = 0.022$) was an order of magnitude lower. Those results evidence the effectiveness of RBA for early detection of refactoring faults.

Regarding RBA, there is an outlier case: the pull up field fault in EasyAccept. In this case, RBA placed the first fault-revealing test cases at position 39, which is far distant from other numbers obtained by RBA. From a detailed analysis, it can be observed that, in this case, the field under refactoring was extensively accessed by several test cases, although the single failing test case was the only one to use test data that revealed the fault. Thus, as the current version of RBA promotes coverage of affected locations for prioritization, disregarding test data, the prioritization process ended up prioritizing other tests over the failing test. As future work, we plan to extend RFMs to include information regarding test data.

There were a few cases in which RBA was outperformed by other techniques; for instance, the pull up field and rename method faults in JMock. In the first case, the failing test case has a high coverage, which favours the *Total* strategies. In the second, the *changed blocks* prioritization performs better, because, by renaming a particular method, the fault was detected by test cases that directly call the changed parts of the code, which is the prioritization heuristic applied by CB. Nevertheless, this result substantiates RBA's stability: even when RBA did not produce the best results, these were quite comparable with the best.

6.2. *Experimental study*

In the second investigation, a controlled experimental study was performed to collect statistical evidence regarding RBA and its prioritization capability.

Questions and metrics This study was conducted based on two research questions:

RQ1: *Can RBA detect refactoring-related faults earlier, when compared with other techniques?*

DOI: 10.1002/stvr

RQ2: *Does RBA place failing test cases in less spread positions than other prioritization techniques?*

For addressing the first question, the F-measure metric was used. Although APFD is the most used for evaluating prioritized suites, this metric is not considered here, for the sake of space and simplicity. As each experiment deals with one refactoring edit and one behavioural change at time, in the scope of this investigation, the APFD results would only reflect the relative magnitude of the F-measure values. Still, all APFD values were calculated and made available at our website [49].

Besides help detecting faults as soon as possible, effective prioritization should place fault-revealing test cases in closer positions. Therefore, to address RQ2, a new metric is proposed, *F-spreading*, which is a rate that measures how the failing test cases are spread in a prioritized test suite. Equation 2 formalizes the metric, where N is the number of test cases of the test suite; m is the number of failing test cases; TF is a sequence containing the positions of failing test cases; and TF_i is the position of the i^o failing test case in the prioritized suite.

$$F\text{-}spreading = \left(\sum_{i=2}^{m} TF_i - TF_{i-1}\right) * \frac{1}{N} \tag{2}$$

Even when a single behavioural change is introduced, several test cases might fail. However, a single test case seldom provides sufficient information for helping fault localization [54]. Particularly, Yoo *et al*. [41] suggest that, for effective fault localization, the next test case to run when a test case fails, should provide as much additional information as possible on the fault locality. In this sense, when failing test cases are narrowly spread, it might easy fault localization, even though we cannot guarantee the ordering always provides maximum information for fault localization. The higher the F-spreading, the more spread the behavioural revealing test cases are.

In summary, good prioritization should generate prioritized suites with low *F-measure* and *F-spreading*. For instance, consider two prioritization techniques *T1* and *T2*, and a test suite *S* with 200 test cases, from which five fail. Suppose that, after applying both *T1* and *T2* to *S*, the failing test cases are placed in the following positions: S_{T1}:{1, 30, 40, 75, 100}, and S_{T2}:{1, 10, 11, 15, 30}. The F-spreading values for S_{T1} and S_{T2} are 0.495 and 0.145, respectively. Although both suites are able to detect the behavioural change early (both $F\text{-}measure_{T1}$ and $F\text{-}measure_{T2}$ are one), *T2* yields a less spread sequence. Thus, by using *T2*, in this context, useful information is confined to a small set of tests, helping to locate the refactoring problem.

Planning and design Regarding the data set, it is difficult to find available real systems in which refactoring behavioural changes can be localized through failing test cases—it is a common police not to commit code with broken tests. Moreover, to the best of our knowledge, there is a lack of refactoring-oriented mutation operators. Therefore, in the context of this experiment, a data set with 26 examples of subtle Java code transformations reported in the literature [48, 55–57] was built. These are the experimental subjects. All those code transformations are free of compilation errors, containing behavioural changes that even well-known refactoring tools (e.g. Eclipse, Netbeans, JRRT) were not able to detect. Figure 10 shows an example of a pull up method edit from the used data set. Method `B.k(int i)` is pulled up to class `A`, generating an unexpected behavioural change. Method `B.test` returns a different result (10, considering the source version, and 20 for the target version). This kind of behaviuoral change is usually hard to identify through visual inspection and was not detected by refactoring tools.

To evaluate consistently the prioritization order produced by each technique, *an extensive test suite was generated for each code transformation*. For that, a test case generation tool was chosen according to the following criteria: (i) the tool should be able to generate a test suite that detects the faults; (ii) the tool should be able to generate more than one test case that fail to suit the practical case that, very often, particularly in a manually created suit, more than one test case fails for a given fault; (iii) the tool should follow a random generation approach to avoid any bias in resemblance of the generation technique and the prioritization heuristics considered by the techniques under evaluation; (iv) the tool should generate white-box test cases to more thoroughly investigate code structural

DOI: 10.1002/stvr

```
1  public class A {
2      public int k (long i){
3          return 10;
4      }
5  }
6  public class B extends A {
7      public int k (int i){
8          return 20;
9      }
10     public int test(){
11         return new A().k(2);
12     }
13 }
```

(a) Original code.

```
1  public class A {
2      public int k (long i){
3          return 10;
4      }
5  +    public int k (int i){
6  +        return 20;
7  +    }
8  }
9  public class B extends A {
10     public int test(){
11         return new A().k(2);
12     }
13 }
```

(b) Code after pulling up method `B.k(int)`.

Figure 10. One pull up method transformation used in the experiment.

Table VI. Hypotheses.

H0: $F\text{-}Measure_{TSC} = F\text{-}Measure_{TMC} = F\text{-}Measure_{ASC} = F\text{-}Measure_{AMC} = F\text{-}Measure_{RD} = F\text{-}Measure_{CB} = F\text{-}Measure_{RBA}$
H1: $F\text{-}Measure_{TSC} \neq F\text{-}Measure_{TMC} \neq F\text{-}Measure_{ASC} \neq F\text{-}Measure_{AMC} \neq F\text{-}Measure_{RD} \neq F\text{-}Measure_{CB} \neq F\text{-}Measure_{RBA}$
H0.2: $F\text{-}Spreading_{TSC} = F\text{-}Spreading_{TMC} = F\text{-}Spreading_{ASC} = F\text{-}Spreading_{AMC} = F\text{-}Spreading_{RD} = F\text{-}Spreading_{CB} = F\text{-}Spreading_{RBA}$
H1.2: $F\text{-}Spreading_{TSC} \neq F\text{-}Spreading_{TMC} \neq F\text{-}Spreading_{ASC} \neq F\text{-}Spreading_{AMC} \neq F\text{-}Spreading_{RD} \neq F\text{-}Spreading_{CB} \neq F\text{-}Spreading_{RBA}$

aspects that are commonly related to refactoring; and (v) the tool should not apply selection/minimization/optimization strategies that could also bias results towards one of the techniques under evaluation.

Consequently, the Randoop tool[¶¶] was selected, because it meets all criteria. Randoop is an automatic unit test generator for Java that produces unit tests by employing feedback-directed random test generation. Moreover, this tool can build regression test suites with no need of user input, having been used in several research studies (e.g. [18, 43, 58, 59]). For instance, the SafeRefactor tool [60, 61] uses Randoop test suites for validating refactorings. It is important to remark that, even though there are other relevant test case generation tools presented in the literature, they may not suit this study context by failling to meet one or more of the aforementioned criteria. For instance, Evosuite [62] does not meet criteria (iii) and (v).

As the size of the code transformations from the data set considered is often small (like the example in Figure 10), and to allow Randoop to generate diversified regression suites, 30 extra methods were added to each subject from the dataset. Those methods are completely independent, not interfering with the execution neither of the original methods, nor of any other extra method. Code for these extra methods is available at [49]; they were not impacted by any of the applied refactorings. Randoop's generation technique combines calls to both original and extra methods into each test case, enriching the test suites. On average, for each of the 26 subjects, Randoop generated suites with an average of 3568 test cases. The same Randoop configuration was used for all generations, with 100 s as time limit, and maximum test size of five statements.

For statistical analysis, two pairs of statistical hypotheses were postulated, null and alternative (Table VI). The null hypotheses (H0 and H0.2) state there is no significant difference between the prioritization techniques under investigation, regarding F-measure and F-spreading (respectively). The alternative hypotheses (H1 and H1.2) state there is significant difference. *One-factor-and-several-treatments* experimental design was applied [63] to each experiment—an experiment for each type of refactoring was considered—where the factor is the prioritization technique and the treatments are seven prioritization techniques (TSC, TMC, ASC, AMC, RD, CB and RBA).

[¶¶] http://randoop.github.io/randoop/

DOI: 10.1002/stvr

Following Jain's suggestion [63], a pilot study was performed for defining the required number of replications for each configuration. For each experimental configuration, the number of replications varied from 500 to 1892, for a precision (r) of 2% of the sample mean and significance (α) of 5%. An experimental configuration is a combination of an object, a prioritization technique and one of the two metrics.

Operation All prioritization executions were performed with the PriorJ tool [64, 65]. PriorJ is an open-source tool that supports test coverage and prioritization activities execution for Java/JUnit systems. PriorJ was extended in order to give support to RBA. Additionally, a set of script classes were written for translating PriorJ's output artefacts and calculating the needed metrics (*F-measure* and *F-spreading*). During the execution of this study, PriorJ was run in a MacBook Pro Core i5 2.4GHz and 4GB RAM, running Mac OS 10.8.4.

Data analysis and discussion First, a normality test was performed, with confidence level of 95% ($\alpha = 0.05$), for each of the 364 experimental configurations (26 Java transformations $\times$ 7 prioritization techniques $\times$ 2 metrics = 364 normality tests). All p-values from those tests were smaller than the significance threshold. Thus, the samples do not follow a normal distribution, and consequently, a non-parametric test should be applied to evaluate the statistical hypothesis. Because each experimental design had a unique factor with more than two treatments, the Kruskal–Wallis test was applied [66]. Again, for all cases, the p-values were smaller than the significance level ($\alpha < 0.05$). Thus, both null hypotheses were rejected, that is, for F-measure and F-spreading, the prioritization techniques presented differences in results, with 95% confidence level.

In a second moment, the confidence intervals for each group of F-measure and F-spreading results were plotted. When overlappings were found, the Mann–Whitney test was applied [66], to pair up and rank the techniques. After analysing confidence intervals and test results, the ranking was established in Table VII. Each row shows a total order of results, for each of the two metrics (columns)—lower to higher metric results, from left to right. For instance, considering the first seeded change with move method (MM_1), RBA was the technique that, on average, had the best F-measure—the technique that leads to detection of this behavioural change earlier. The second best technique for this scenario was either RD or CB (their results were statistically similar), then TMC or AMC. Finally, TSC and ASC produced the highest F-measure, resulting in the worst order for detecting the seeded fault. More detailed information regarding this analysis (normality and hypothesis tests) is available in [49].

The results evidence that RBA promotes early detection of refactoring-related behavioural changes. With respect to F-measure, RBA performed better than, or at least similar to, the other techniques, for all configurations and refactoring types. Even CB did not behave well in several cases (e.g. MM_4, MM_5). Regarding RQ1, RBA showed to be the better choice.

In addition, concerning F-spreading, RBA presented the best numbers in placing failing test cases in close positions. Thus, RQ2 can be answered by suggesting that RBA very often places behavioural-change-revealing test cases close to one another. This fact may give testers/developers confined information to support fault localization.

By examining the cases, RBA was outperformed (MM_6, PUF_1, PUF_2, PUF_3), and it can be observed that not all methods collected by RFMs were impacted by the change. This happened because the RFMs are name based, which may not be efficient in some situations. In specific, when the refactoring edit involves variable manipulation (e.g. pull up field), name analysis is compromised by variable with same name but different scopes, as happened in this experiment. For instance, when a name-based approach was used for searching for methods that accessed certain refactored field (rule from the pull up field's RFM), by coincidence, some of the extra methods had local variables with same name as refactored fields. Thus, for those cases, this analysis failed on selecting only the test cases related to the changes.

In order to explore this supposition, a post-study investigation was performed, renaming those variables before rerunning the experiment. As result, there was a significant improvement of the F-spreading results. For instance, PUF_1's F-spreading dropped from 0.95 to just 0.004. These numbers may be evidence that, by combining the proposed RFM rules with variable scope

DOI: 10.1002/stvr

Table VII. Results of prioritization techniques, for each type of refactoring.

Move Method

		F-Measure	F-Spreading
(a)	MM_1	RBA < (RD = CB) < (TMC = AMC) < (TSC = ASC)	RBA < (TSC = ASC) < (TMC = AMC) < (CB = RD)
	MM_2	RBA = CB < RD < (TMC = AMC) < (TSC = ASC)	*
	MM_3	RBA < RD < CB < (TMC = AMC) < (TSC = ASC)	*
	MM_4	RBA < (TMC = AMC) < (CB= RD) < (TSC = ASC)	RBA < CB < (RD = TSC = ASC) < (TMC = AMC)
	MM_5	RBA < (TMC = AMC) < (RD = CB) < (TSC = ASC)	RBA < (TMC = AMC) < ASC < TSC < (CB = RD)
	MM_6	(RBA = CB) < RD < (TMC = AMC) < (TSC = ASC)	CB < RBA < (TMC = AMC) < (TMC = AMC) < RD
	MM_7	RBA < (RD = CB) < (TMC = AMC) < (TSC = ASC)	RBA < (TSC = ASC) < (TMC = AMC) < (CB = RD)
	MM_8	RBA < (CB = RD) < (TMC = AMC) < (TSC = ASC)	RBA < (TSC = ASC) < (TMC = AMC) < (CB = RD)

Rename Method

		F-Measure	F-Spreading
(b)	RM_1	RBA (CB = RD) < (TMC = AMC) < (TSC = ASC)	RBA < (TMC = AMC) < (TSC = ASC) < (CB = RD)
	RM_2	(RBA = CB = RD) < (TMC = AMC =TSC = ASC)	*

Pull Up Field

		F-Measure	F-Spreading
(c)	PUF_1	(RBA = CB = RD) < (TMC = AMC) < (TSC = ASC)	(TSC = ASC) < (TMC = AMC) < (RBA = CB = RD)
	PUF_2	(RBA = CB = RD) < (TMC = AMC) < (TSC = ASC)	(TSC = ASC) < (TMC = AMC) < (RBA = CB = RD)
	PUF_3	(RBA = CB = RD) < (TMC = AMC) < (TSC = ASC)	(TSC = ASC) < (TMC = AMC) < (RBA = CB = RD)

Pull Up Method

		F-Measure	F-Spreading
(d)	PUM_1	RBA < (CB = RD) < (TMC = AMC) < (TSC = ASC)	RBA < (TMC = AMC) < (TSC = ASC) < (CB = RD)
	PUM_2	(RBA = CB) < (TMC = AMC) < RD < (TSC = ASC)	(RBA = CB) < (TMC = AMC) < (TSC = ASC) < RD
	PUM_3	RBA < RD < (TMC = AMC) < CB < (TSC = ASC)	(RBA = CB) < (TSC = ASC)< (TMC = AMC) < RD
	PUM_4	(RBA = CB) < (TMC = AMC) < RD < (TSC = ASC)	(RBA = CB) < (TMC = AMC) < (TSC = ASC) < RD
	PUM_5	(RBA = CB) < (TMC = AMC) < RD < (TSC = ASC)	(RBA = CB) < (TMC = AMC)< (TSC = ASC) < RD
	PUM_6	RBA < (TMC = AMC) < (CB = RD) < (TSC = ASC)	RBA < (TMC = AMC) < (TSC = ASC) < (RD = CB)
	PUM_7	RBA < (CB = RD = TMC = AMC) < (TSC = ASC)	RBA < (TSC = ASC) < (TMC = AMC) < (CB = RD)
	PUM_8	(RBA = CB)< (TMC = AMC) < RD < (TSC = ASC)	(RBA = CB) < (TMC = AMC) < (TSC = ASC) < RD
	PUM_9	(RBA = CB) < (TMC = AMC) < RD < (TSC = ASC)	(RBA = CB) < (TMC = AMC) < (TSC = ASC) < RD

Add Parameter

		F-Measure	F-Spreading
(e)	AP_1	RBA < (TMC = AMC) < (RD = CB) < (TSC = ASC)	RBA < (TMC = AMC) < ASC < TSC < (CB = RD)
	AP_2	RBA < (CB = RD) < (TMC = AMC) < (TSC = ASC)	RBA < (TMC = AMC) < (TSC = ASC) < (CB = RD)
	AP_3	(RBA = CB) < RD < (TMC = AMC) < (TSC = ASC)	(RBA = CB) < (TMC = AMC) < (TSC = ASC) < RD
	AP_4	(RBA = CB) < (TMC = AMC) < RD < (TSC = ASC)	(RBA = CB) < (TMC = AMC) < (TSC = ASC) < RD

* Impossible to calculate (single failed test case)

differentiation and/or binding checking, the quality of the prioritization results may improve. A new version of changed RFMs and additional experiments are regarded as future work.

Numerical analysis Beyond the statistical evidence presented earlier, the overall numbers of the experiment emphasize conclusions presented in this paper. Each prioritization technique was run, on average, 3545 times, generating one different prioritized suite for each run. By analysing the values for both metrics (F-measure and F-spreading) are possible to see that in 71% of the cases, RBA yielded better F-measure results, in comparison with all other techniques. Similarly, considering the F-spreading results, RBA outperformed the other techniques in 73% of the cases. The data containing all results from this study are available in [49].

Final remarks The reported experimental study deals with two complex elements: (i) subtle *refactoring faults*, which not even the most well-known refactoring tools are able to detect; and (ii) *large regression test suites*, encompassing an average of 3568.6 JUnit test cases. Those elements tend to turn prioritization even harder. Even though dealing with those complicating factors, RBA produced satisfactory and stable results. With the combination of early fault localization (high F-measure) and a more narrowly spread of failing test cases (low F-spreading), RBA appears as an alternative for test case prioritization when doing refactoring.

Complementary study with evosuite Although experimentation presented in this section is based on test suites generated using the Randoop tool, a pilot study was conducted to investigate weather RBA's results would be different when dealing with suites generated using other strategies. For that, other well-known test generation tool was selected, Evosuite [62]. Evosuite was used in its default configuration for generating suites for the same data set with 26 subjects. The size of the generated

DOI: 10.1002/stvr

test suites were considerably smaller than the ones generated by Randoop (around 27 test cases per suite), as EvoSuite performs optimizations and minimizations. Very often, only one test case failed (only in 4 out of the 26 subject, 2–3 test cases failed), and there was a case where the fault was not detected by the generated suite (RM_2 object). Therefore, with EvoSuite, it was only possible to partially evaluate the techniques by observing their detection potential (F-measure). It was not possible to assess spreading of test cases that fail (F-spreading).

Even so, RBA had a similar and excellent performance, as in the study with Randoop, regarding F-measure, placing the unique test case that fail in the first position in the majority of the cases (80%) and, in the worst case, in the 3rd position. On the other hand, the second best prioritization technique (CB) placed the failed test cases among the three first position in only 34% of the cases. All results of this complementary study are available in our website [49]. It is important to remark that coverage-based techniques could not be considered in this study as, differently from Randoop, Evosuite does not mix the target methods with the additional methods in test cases. As a consequence, the size of the methods can bias the results obtained for these techniques.

Threats to validity Regarding the possible threats to the validity of the results previously discussed:

- **Conclusion**: In order to achieve statistical significance, the number of replications for each experiment was decided according to statistical principles. Moreover, the analysis applied considered a high confidence level (95%). Finally, the statistical tests were selected after testing the data set against normality;
- **Internal**: The existence of potential faults in PriorJ could have undermined results analysis. However, PriorJ contains a set of extensive unit tests that validate how it implements the prioritization techniques. Additionally, for controlling this threat, this tool was validated through testing on several examples of test suites and programs.
- **Construction**: Alternatives to the metrics considered—F-measure and F-spreading—could have been considered for evaluating prioritization. However, according to the purpose of this investigation, and the experimental setup, those measures can be regarded as appropriate. Future additional studies might consider different metrics.
- **External**: Some circumstances of the study certainly hinder the generalization of its results. First, because the number of subjects were small in some cases (e.g. two code examples for the rename method edit), it is impossible to say that the used data set represents the whole universe of Java programs and refactoring faults. However, as those subjects were identified by other research results, and reflect subtle faults that not even the most used refactoring tools were able to identify, they can be regarded as suitable for the purpose of this investigation, based on the assumption that if a prioritization technique is able to detect those hard to find faults, it is likely to detect easier ones. Second, concerning test suite representativeness, only automatically generated random regression suites were used. Although this practice is not always used in real projects, random testing has been used extensively, and random unit tests have been a great alternative due to the available tool support (e.g. Randoop). In addition, random suites have been used by other works for validating refactorings [61]. Moreover, manually created test suites were used in the case study with open source projects (Section 6.1).

7. RELATED WORK

Refactoring and testing Validation of refactoring edits with testing is common practice in real projects. Although other refactoring validation strategies are attainable (e.g. [18, 67]), regression test suites are still the main (mostly the only) strategy for assuring correctness of manual refactorings in daily development. A study with Microsoft developers [4] shows the lack of effective tests often refrains developers from starting to refactor. Moreover, even well-known engines for automatic Java refactoring, such as Eclipse, NetBeans and JRRT, are not exempt from faults, which emphasizes the need for testing validation. Test-based strategies have revealed important bugs in those tools [10, 55].

DOI: 10.1002/stvr

However, as software evolves, its regression suite tends to increase, becoming hard to manage. Nevertheless, as shown by Rachatasumrit and Kim [14], often only part of a suite is useful for revealing refactoring faults. One way of dealing with this limitation is to generate tests automatically, as employed by other tools for validating refactorings, such as SafeRefactor [18] and SafeRefactorImpact [43]. However, it may be impractical to regenerate, or only execute, an extensive test suite after each refactoring. In this sense, by providing a strategy for rescheduling test cases, applicable to any Java test suite—whether it was manually or automatically created—RBA might harmonize with these test-based refactoring validation strategies. Once a suite is generated, its test cases can be rescheduled by RBA and, because the top test cases are more likely to review the faults, a developer can decide how much of the suite should be executed, according to her resource constraints.

Test case prioritization To the best of our knowledge, RBA is the first prioritization approach specialized in early detection of refactoring faults. Still, considerable research has been produced in the past decades on test case prioritization.

Singh *et al*. [68] perform an elaborate literature review that maps the state-of-art of test case prioritization. From an initial set of 12 977 studies, they discuss 106 prioritization techniques split into eight categories: *coverage based*, *modification based*, *fault based*, *requirement based*, *history based*, *genetic based*, *composite approaches* and other approaches. Even though there is a modification-based category, none of the techniques identified by their study emphasize the impact of refactoring changes. A similar conclusion can be extracted from the systematic mapping study performed by Catal and Mishra [69] and from Yoo and Harman's survey [70]. This fact underpins RBA's novelty in underlining refactoring faults. Moreover, the modification-based techniques discussed in those studies require an abstract representation of the program (e.g. finite state machines) before performing their prioritization. Those models are not always available in real projects. RBA, on the other hand, requires only two consecutive versions of a program to be able to perform its prioritization, which tends to be more practical and less costly.

Coverage-based prioritization techniques are simple and frequently used in practice. They base prioritization on the assumption that test cases with high coverage of program statements or methods are more likely to reveal faults. Those techniques have been successfully used to early detect general faults. For instance, Rothermel *et al*. [37] present a set of coverage-based prioritization techniques, which are evaluated over their fault detection rates. The authors show that even the least costly technique significantly improves detection. They also suggest there might be room for improvement in coverage-based prioritization. These conclusions are reassured by previous empirical studies [34–36] in which general-purpose techniques proved inadequate for early detection of refactoring faults; the combination of test coverage and the impact analysis applied by RBA presented significantly better results.

Srivastava and Thiagarajan [38] propose a modification-based prioritization technique that focuses on rescheduling regression tests according to their coverage on code statements modified between two consecutive versions of a program (changed blocks). This technique was used in the empirical studies presented in this paper, labelled as CB. By focusing only on the modified statements, this technique often misses behavioural changes that are not directly associated with these statements; for instance, renaming a method may affect several methods down its class hierarchy. Moreover, they do not target specific types of changes, making it less effective for dealing with refactoring problems.

Regarding prioritization techniques that do not involve refactoring, Zhang *et al*. [71] propose models for unifying the *total* and *additional* strategies, while Jeffrey and Gupta [72] present an algorithm that prioritizes test cases based on their coverage of relevant slices of test outputs, comparing their technique with traditional coverage-based techniques. Similarly, Korel *et al*. [23] propose prioritization techniques based on system models that are associated with code information.

In addition, Srikanth *et al*. [26] propose a prioritization approach based on requirements volatility, customer priority, implementation complexity and fault proneness of requirements, whereas Walcott *et al*. [25] present a prioritization technique that uses genetic algorithms to reorder test suites based on testing time constraints. Park *et al*. [73], in turn, use historical information to estimate the current cost and fault severity for cost-cognizant test case prioritization, and Mei *et al*. [74] propose a static

DOI: 10.1002/stvr

approach for prioritizating JUnit test cases in the absence of coverage information. Finally, Sanchez *et al*. [75] explore the applicability of test case prioritization to software product lines, by proposing five different prioritization criteria based on common metrics from feature models.

Change impact analysis Different from other approaches, RBA performs lightweight, test-focused change impact analysis, emphasizing commonly impacted code declarations, using RFMs. Traditional change impact approaches often use either static or dynamic analysis; in RBA, these two types of analysis are combined.

Bohner and Arnold [76] measure impact through reachability on call graphs. Although intuitive, their strategy may be imprecise, only tracking methods downstream from the changed method. On the other hand, Law and Rothermel [77] propose *PathImpact*, a dynamic impact analysis strategy based on whole-path profiling. From the changed method, *PathImpact* goes back and forth over the execution trace in order to determine the impact after the change. One important issue related to impact analysis is that, in general, when dealing with a major edit (e.g. a structural refactoring), they tend to gather a large number of methods. When relating those impacted methods to regression tests, too many test cases are likely to be selected. RBA reduces this risk by focusing only on methods that a refactoring edit is most likely to affect.

Regarding change-based approaches, Ren *et al*. [78, 79] first decomposes the differences between two versions of a program into atomic changes (e.g. *add a field* and *delete a field*). Then, call graphs from test cases are analysed, based on rules that identify, from the suite, the subset potentially affected by the changes. RBA employs a similar approach for detecting the possibly affected test cases by also using call graphs. RFMs, however, are the basis of RBA's impact analysis. In contrast, Ren *et al*. approach does not distinguish changes by the type of edit. Furthermore, by working at a different granularity level—refactoring edits, rather than atomic changes—RBA tends to select a smaller group of test cases. Although this set might be not complete, RBA experimentally outperforms other prioritization techniques.

Zhang *et al*. [80, 81] extend the work of Ren *et al*. by improving its impact analysis, plus a spectrum analysis that helps change inspection. For that, they rank test cases according to their likelihood of localizing faults. Their results evidence the importance of helping fault localization, which can also be a benefit of RBA. RBA avoids spectrum analysis as the test cases directly related to the common refactoring faults are selected. Therefore, the test cases more likely to reveal those problems are placed on top positions of the prioritized suite, and also close to one another.

Wloka *et al*. [82] propose an approach that employs change impact analysis to guide developers in creating new unit tests. This analysis identifies code changes not covered in the current test suite, and indicating whether tests miss those changes. Results are then presented, and the developer has the option to extend the suite to cover the problematic locations. Differently, from RBA, this approach is fine grained—an edit is decomposed into atomic changes—which might generate a bigger affected set than the one collected by RFMs. Moreover, their approach goes towards test augmentation, which is not the focus of the work presented in this paper. Rather the focus is on projects with massive test suites, in need to early fault detection. Mongiovi *et al*. [43] propose SafeRefactorImpact, a tool that adds impact analysis to the SafeRefactor validation tool [18]. SafeRefactorImpact detects behaviour-changing transformations in both object and aspect-oriented programs. For that, it decomposes an edit into fine-grained changes then finds a set of affected methods for each atomic change. Next, it applies Randoop to generate suites that are run against the two versions of the program. SafeRefactorImpact and RFMs are similar in the sense that both focus on methods that are commonly impacted by a refactoring edit. In contrast, RBA differs in granularity; by focusing on the refactoring edit as a single object, RFMs select less test cases. Moreover, RBA is designed to promote early detection of refactoring faults in a sound test suite, while SafeRefactorImpact generates new tests for that focus on detecting those faults. In this context, both approaches can definitely be used in combination.

Call graph generation Call graphs can be either dynamically or statically generated, although the latter may miss accurate information related to subtyping and dynamic dispatch. There are several tools that automatically generate call graphs for several programming languages

DOI: 10.1002/stvr

(e.g. PriorJ [64], KCachegrind [83] and PhpcallgraphSite [84]). The implementation of RBA, in the context of PriorJ, uses dynamic call graphs.

8. CONCLUSION

This article presents the RBA a test case prioritization approach that aims to promote early detection of behavioural changes introduced after refactorings. RBA first identifies the edits performed between two versions of a program, collects methods that might have been affected by the change—by applying a lightweight impact analysis based on a set of RFMs—and reorders the regression test cases according to their coverage of those methods. The RFMs summarize common methods that may be impacted by a specific refactoring type. RBA was evaluated regarding effectiveness through a case study and experimental studies. The case study shows the applicability of RBA in real projects when dealing with real test suites. The results of experimental studies show statistical evidence that, in fact, RBA promotes early detection of refactoring problems, when compared with other prioritization techniques. Moreover, RBA tends to place fault-revealing test cases in closer positions, which can be very helpful for pinpointing the fault.

Refactoring-based approach , as it uses coverage data, demands at least one complete execution of the test suite to generate call-graphs. One may argue this limitation weakens the cost-benefit ratio of using the approach. In fact, running the entire suite may be costly; still, refactoring validation demands this action anyway, probably more than once. In addition, the ordering provided by RBA with a more narrowly spread of failing test cases may save time during fault localization. Finally, test cases that become obsolete can also be identified and possibly discarded. RBA advances the state-of-the-art of test prioritization focusing on refactoring faults. Particularly, RBA leverages developer's reliance on test suites for detecting refactoring problems, by proposing distinct execution orders that speed up this process. Unlike other prioritization techniques, RBA's prioritization strategy is specialized for each type of refactoring, according to its mechanics and commonly impacted methods. RBA's solution does not affect the suite's fault detection potential, as no test case is discarded, and the number of test cases to execute can be tuned based on available resources.

As future work, RFMs will be extended. New rules regarding variable reference checking and/or test data variability will be included. This extension will reduce the rate of false positives in the affected method set. Additionally, RFMs for other well-known refactorings will be developed, such as extract method and decompose conditional. Moreover, there is a lack of refactoring-oriented mutation operators. In order to help the evaluation of refactoring validation strategies and to minimize human bias, refactoring-based mutation operators for Java systems will be developed. Finally, RBA's evaluation will be extended by better investigating the use of RBA under the context of automatically generated test suites. Therefore, a number of different test generation tools will be considered.

REFERENCES

1. Fowler M, Beck K. *Refactoring: Improving The Design of Existing Code*. Addison-Wesley Professional: Boston, 1999.
2. Mens T, Tourwé T. A survey of software refactoring. *IEEE Transactions on Software Engineering* 2004; **30**(2): 126–139.
3. Xing Z, Stroulia E. Umldiff: an algorithm for object-oriented design differencing. *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ACM: Long Beach, 2005; 54–65.
4. Kim M, Zimmermann T, Nagappan N. A field study of refactoring challenges and benefits. *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ACM: Cary, 2012; 50.
5. MacCormack A, Rusnak J, Baldwin C Y. Exploring the structure of complex software designs: an empirical study of open source and proprietary code. *Management Science* 2006; **52**(7):1015–1030.
6. Murphy GC, Kersten M, Findlater L. How are Java software developers using the elipse IDE *Software, IEEE* 2006; **23**(4):76–83.
7. Negara S, Chen N, Vakilian M, Johnson RE, Dig D. *A Comparative Study of Manual and Automated Refactorings. ECOOP 2013—Object-Oriented Programming*. Springer: Berlin, 2013.
8. Vakilian M, Chen N, Negara S, Rajkumar BA, Bailey BP, Johnson RE. Use, disuse, and misuse of automated refactorings. *2012 34th International Conference on Software Engineering (ICSE)*, IEEE: Zurich, 2012; 233–243.

DOI: 10.1002/stvr

9. Lee YY, Chen N, Johnson RE. Drag-and-drop refactoring: intuitive and efficient program transformation. *Proceedings of the 2013 International Conference on Software Engineering*, IEEE Press: San Francisco, 2013; 23–32.
10. Daniel B, Dig D, Garcia K, Marinov D. Automated testing of refactoring engines. *Proceedings of the The 6th Joint Meeting of the European Software Engineering Conference and The ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ACM: Dubrovnik, 2007; 185–194.
11. Soares G, Gheyi R, Massoni T. Automated behavioral testing of refactoring engines. *IEEE Transactions on Software Engineering* 2013; **39**(2):147–162.
12. Dig D, Johnson R. The role of refactorings in api evolution. *Proceedings of the 21st IEEE International Conference on Software maintenance, 2005. ICSM'05*, IEEE: Budapest, 2005; 389–398.
13. Weißgerber P, Diehl S. Are refactorings less error-prone than other changes?. *Proceedings of the 2006 International Workshop on Mining Software Repositories*, ACM: Shanghai, 2006; 112–118.
14. Rachatasumrit N, Kim M. An empirical investigation into the impact of refactoring on regression testing. *2012 28th IEEE International Conference on Software Maintenance (ICSM),* Trento, 2012; 357–366.
15. Cornélio M, Cavalcanti A, Sampaio A. Sound refactorings. *Science of Computer Programming* 2010; **75**(3):106–133.
16. Mens T, Van Gorp P. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science* 2006; **152**:125–142.
17. Overbey JL, Fotzler MJ, Kasza AJ, Johnson RE. A collection of refactoring specifications for Fortran 95. *ACM SIGPLAN Fortran Forum*, Vol. 29, ACM: New York, 2010; 11–25.
18. Soares G, Gheyi R, Serey D, Massoni T. Making program refactoring safer. *Software, IEEE* 2010; **27**(4):52–57.
19. Leung HK, White L. Insights into regression testing [software testing]. *Software Maintenance, 1989, Proceedings, Conference on*, IEEE: Miami, 1989; 60–69.
20. Stoerzer M, Ryder BG, Ren X, Tip F. Finding failure-inducing changes in Java programs using change classification. *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM: Portland, 2006; 57–68.
21. Rothermel G, Untch R, Chu C, Harrold M. Prioritizing test cases for regression testing. *Software Engineering, IEEE Transactions on* 2001; **27**(10):929–948.
22. Wong W, Horgan J, London S, Agrawal H. A study of effective regression testing in practice. *PROCEEDINGS The Eighth International Symposium On Software Reliability Engineering*, IEEE: Albuquerque, 1997; 264–274.
23. Korel B, Tahat L, Harman M. Test prioritization using system models. *Proceedings of the 21st IEEE International Conference on Software Maintenance, 2005. ICSM'05*, IEEE: Budapest, 2005; 559–568.
24. Li Z, Harman M, Hierons R. Search algorithms for regression test case prioritization. *Software Engineering, IEEE Transactions on* 2007; **33**(4):225–237.
25. Walcott KR, Soffa ML, Kapfhammer GM, Roos RS. TimeAware test suite prioritization. *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, *(ISSTA '06)*, Portland, 2006; 1–12.
26. Srikanthm H, Williams L, Osborne J. System test case prioritization of new and regression test cases. *2005 International Symposium on Empirical Software Engineering, 2005,* Queensland, 2005; 10.
27. Rummel MJ. Towards the prioritization of regression test suites with data flow information. *In Proceedings of the 20th Symposium on Applied Computing*, ACM Press: Santa Fe, 2005.
28. Ramanathan MK, Koyuturk M, Grama A, Jagannathan S. Phalanx: a graph-theoretic framework for test case prioritization. *Proceedings of the 2008 ACM Symposium on Applied Computing, (SAC) '08*, ACM: New York, NY, USA, 2008; 667–673.
29. Kazarlis SA, Bakirtzis A, Petridis V. A genetic algorithm solution to the unit commitment problem. *Power Systems, IEEE Transactions on* 1996; **11**(1):83–92.
30. Mei L, Zhang Z, Chan W K, Tse T H. Test case prioritization for regression testing of service-oriented business applications. *Proceedings of the 18th International Conference on World Wide Web, (WWW '09)*, ACM, 2009; 901–910.
31. Mei L, Chan WK, Tse T, Merkel RG. XML-manipulating test case prioritization for XML-manipulating services. *Journal of Systems and Software* 2011; **84**(4):603–619.
32. Rothermel G, Harrold M J, Dedhia J. Regression test selection for C++ software. *Software Testing Verification and Reliability* 2000; **10**(2):77–109.
33. Ruth M, Tu S. A safe regression test selection technique for web services. *Second International Conference on Internet and Web Applications and Services, 2007. (ICIW'07)*, IEEE: Mauritius, 2007; 47–47.
34. Alves EL, Machado PD, Massoni T, Santos ST. A refactoring-based approach for test case selection and prioritization. *Automation of Software Test (AST), 2013 8th International Workshop on*, IEEE: San Francisco, 2013; 93–99.
35. Alves ELG. Investigating test case prioritization techniques for refactoring activities validation: evaluating the behavior, technical report. *Technical Report SPLab-2012-001*, Software Practices Laboratory, UFCG, May 2012. http://splab.computacao.ufcg.edu.br/technical-reports/, http://splab.computacao.ufcg.edu.br/technical-reports/.
36. Alves ELG. Investigating test case prioritization techniques for refactoring activities validation: evaluating suite characterists. Technical report. *Technical Report SPLab-2012-002*, Software Practices Laboratory, August 2012. http://splab.computacao.ufcg.edu.br/technical-reports/, http://splab.computacao.ufcg.edu.br/technical-reports/.
37. Rothermel G, Untch R, Chu C, Harrold M. Test case prioritization: an empirical study. Software maintenance. *Proceedings of IEEE International Conference on Software maintenance (ICSM'99)*, IEEE: Oxford, 1999; 179–188.

DOI: 10.1002/stvr

38. Srivastava A, Thiagarajan J. Effectively prioritizing tests in development environment. *ACM SIGSOFT Software Engineering Notes*, Vol. 27, ACM: New York, 2002; 97–106.
39. Jiang B, Zhang Z, Tse T, Chen T. How well do test case prioritization techniques support statistical fault localization. *33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC '09)*, Vol. 1, Seattle, 2009; 99–106.
40. Gonzalez-Sanchez A, Piel E, Gross HG, van Gemund A. Prioritizing tests for software fault localization. *2010 10th International Conference on Quality Software (QSIC)*, 2010; 42–51.
41. Yoo S, Harman M, Clark D. Fault localization prioritization: comparing information-theoretic and coverage-based approaches 2013; **22**(3):19:1–19:29.
42. Xia X, Gong L, Le TD, Lo D, Jiang L, Zhang H. Diversity maximization speedup for localizing faults in single-fault and multi-fault programs. *Automated Software Engineering* 2014:1–33.
43. Mongiovi M, Gheyi R, Soares G, Teixeira L, Borba P. Making refactoring safer through impact analysis. *Science of Computer Programming* 2014; **93**:39–64.
44. Elbaum S, Malishevsky A, Rothermel G. Test case prioritization: a family of empirical studies. *IEEE Transactions on Software Engineering* 2002; **28**(2):159–182.
45. Murphy-Hill E, Parnin C, Black AP. How we refactor, and how we know it. *IEEE Transactions on Software Engineering* 2012; **38**(1):5–18.
46. Kim M, Gee M, Loh A, Rachatasumrit N. Ref-finder: a refactoring reconstruction tool based on logic query templates. *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ACM: Santa Fe, 2010; 371–372.
47. Grove D, DeFouw G, Dean J, Chambers C. Call graph construction in object-oriented languages. *ACM SIGPLAN Notices* 1997; **32**(10):108–124.
48. Soares G. Automated behavioral testing of refactoring engines. *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, ACM: Tucson, 2012; 49–52.
49. Rba website. https://sites.google.com/a/computacao.ufcg.edu.br/rba/. [last accessed 27 July 2014].
50. Naumann DA, Sampaio A, Silva L. *Refactoring and representation independence for class hierarchies*, 2012.
51. Pit website. http://pitest.org/. [last accessed 3 February 2014].
52. Chen T, Leung H, Mak I. Adaptive random testing. In *Advances in Computer Science—ASIAN 2004. Higher-level Decision Making, Lecture Notes in Computer Science*, Vol. 3321, Maher M (ed.) Springer: Berlin Heidelberg, 2005; 320–329.
53. Jiang B, Zhang Z, Chan W, Tse T. Adaptive random test case prioritization. *24th IEEE/ACM International Conference on Automated Software Engineering (ASE'09)*, IEEE: Auckland, 2009; 233–244.
54. Jones JA, Harrold MJ. Empirical evaluation of the tarantula automatic fault-localization technique. *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, (ASE '05)*, ACM: Long Beach, 2005; 273–282.
55. Soares G, Gheyi R, Massoni T. Automated behavioral testing of refactoring engines. *IEEE Transactions on Software Engineering* 2013; **39**(2):147–162.
56. Gligoric M, Gvero T, Jagannath V, Khurshid S, Kuncak V, Marinov D. Test generation through programming in udita. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-*, Vol. 1, ACM: Cape Town, 2010; 225–234.
57. Daniel B, Dig D, Garcia K, Marinov D. Automated testing of refactoring engines. *ESEC/FSE 2007: Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM Press: New York, NY, USA, 2007.
58. Daniel B, Boshernitsan M. Predicting effectiveness of automatic testing tools. *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, IEEE: L'Aquila, 2008; 363–366.
59. Inkumsah K, Xie T. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, IEEE: L'Aquila, 2008; 297–306.
60. Saferefactor website. http://www.dsc.ufcg.edu.br/spg/saferefactor/. [last accessed 31 August 2015].
61. Soares G, Cavalcanti D, Gheyi R, Massoni T, Serey D, Cornélio M. Saferefactor-tool for checking refactoring safety. *Tools Session at SBES* 2009:49–54.
62. Fraser G, Arcuri A. Evosuite: automatic test suite generation for object-oriented software. *9th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ACM: New York, 2011; 416–419.
63. Jain R. *The Art of Computer Systems Performance Analysis*, Vol. 182. John Wiley & Sons: Chichester, 1991.
64. Rocha J, ELG A, Machado P. Priorj—priorizao automatica de casos de teste junit. *Proceedings of Third Brazilian Conference on Software: Theory and Practice (CBSoft)—Tools Section* 2012; **4**:43–50.
65. Alves E, Santos T, Machado PTM. Test case prioritization by using PriorJ. *Proceedings of 7th Brazilian Workshop on Systematic and Automated Software Testing (SAST, 2013),* Brasilia, 2013.
66. Wohlin C, Runeson P, Höst M, Ohlsson M C, Regnell B, Wesslén A. *Experimentation in Software Engineering*. Springer, 2012.
67. Ge X, Murphy-Hill E. Manual refactoring changes with automated refactoring validation. *Proceedings of the 36th International Conference on Software Engineering*, ACM: Hyderabad, 2014; 1095–1105.
68. Singh Y, Kaur A, Suri B, Singhal S. Systematic literature review on regression test prioritization techniques. In *Special Issue: Advances in Network Systems Guest Editors*, Vol. 379, Andrzej Chojnacki (ed.)

DOI: 10.1002/stvr

69. Catal C, Mishra D. Test case prioritization: a systematic mapping study. *Software Quality Journal* 2013; **21**(3): 445–478.
70. Yoo S, Harman M. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability* 2012; **22**(2):67–120.
71. Zhang L, Hao D, Zhang L, Rothermel G, Mei H. Bridging the gap between the total and additional test-case prioritization strategies. *2013 35th International Conference on Software Engineering (ICSE)*, IEEE: San Francisco, 2013; 192–201.
72. Jeffrey D, Gupta R. Test case prioritization using relevant slices. 30th Annual International Computer Software and Applications Conference (COMPSAC'06), Vol. 1, IEEE: Chicago, 2006; 411–420.
73. Park H, Ryu H, Baik J. Historical value-based approach for cost-cognizant test case prioritization to improve the effectiveness of regression testing. *Second International Conference on Secure System Integration and Reliability Improvement (SSIRI'08)*, IEEE, 2008; 39–46.
74. Mei H, Hao D, Zhang L, Zhang L, Zhou J, Rothermel G. A static approach to prioritizing junit test cases. *IEEE Transactions on Software Engineering* 2012; **38**(6):1258–1275.
75. Sánchez A B, Segura S, Ruiz-Cortés A. A comparison of test case prioritization criteria for software product lines. *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation (ICST)*, IEEE: Cleveland, 2014; 41–50.
76. Bohner SA, Arnold RS. Software change impact analysis, chap. *An Introduction to Software Change Impact Analysis* 1996; **1**:1–26.
77. Law J, Rothermel G. Whole program path-based dynamic impact analysis. Software engineering, 2003. *Proceedings of the 25th International Conference on Software Engineering*, IEEE: Portland, 2003; 308–318.
78. Ren X, Shah F, Tip F, Ryder B, Chesley O. Chianti: a tool for change impact analysis of Java programs. *ACM SIGPLAN Notices*, Vol. 39, ACM: New York, 2004; 432–448.
79. Ren X, Ryder B, Stoerzer M, Tip F. Chianti: a change impact analysis tool for Java programs. *Proceedings of the 27th International Conference on Software Engineering*, IEEE: St. Louis, 2005; 664–665.
80. Zhang L, Kim M, Khurshid S. Localizing failure-inducing program edits based on spectrum information. *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, IEEE: Williamsburg, 2011; 23–32.
81. Zhang L, Kim M, Khurshid S. *Faulttracer: A change impact and regression fault analysis tool for evolving Java programs*, 2012.
82. Wloka J, Hoest E, Ryder B. Tool support for change-centric test development. *Software, IEEE* 2010; **27**(3):66–71.
83. Kcachegrind website. http://kcachegrind.sourceforge.net/. [last accessed 31 August 2015].
84. Phpcallgraph website. http://phpcallgraph.sourceforge.net/. [last accessed 31 August 2015].

DOI: 10.1002/stvr