A Refactoring-Based Approach for Test Case Selection and Prioritization

Everton L. G. Alves, Patricia D. L. Machado, Tiago Massoni, and Samuel T. C. Santos
SPLab - Software Practices Laboratory, Federal University of Campina Grande, UFCG, Campina Grande, Brazil
everton@copin.ufcg.edu.br, {patricia,massoni}@computacao.ufcg.edu.br, samuel.santos@ccc.ufcg.edu.br

Abstract—**Refactoring edits, commonly applied during software development, may introduce faults in a previously-stable code. Therefore, regression testing is usually applied to check whether the code maintains its previous behavior. In order to avoid rerunning the whole regression suite, test case prioritization techniques have been developed to order test cases for earlier achievement of a given goal, for instance, improving the rate of fault detection during regression testing execution. However, as current techniques are usually general purpose, they may not be effective for early detection of refactoring faults. In this paper, we propose a refactoring-based approach for selecting and prioritizing regression test cases, which specializes selection/prioritization tasks according to the type of edit made. The approach has been evaluated through a case study that compares it to well-known prioritization techniques by using a real open-source Java system. This case study indicates that the approach can be more suitable for early detection of refactoring faults when comparing to the other prioritization techniques.**

I. Introduction

During software development and maintenance, one of the most common activities performed is refactoring [22], [10], [11]. A refactoring edit is a kind of code transformation aiming at performing structural adjustments in the source code without adding or removing any behavioral feature. Those adjustments are usually related to software quality improvements (*e.g.* maintainability, readability). Recent studies have shown that nearly 30% of the edits performed during a software development are likely to be refactorings [28]. Especially when development teams are guided by an agile methodology (*e.g.* XP [4]), this number tends to increase.

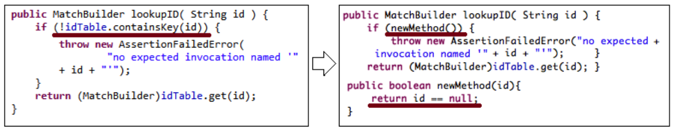
Refactoring edits can vary from a simple variable renaming to more complex changes (*e.g.* changes in hierarchical relationships). Independently of the scope of the edit, a refactoring process is usually very hard to manage. Several subtle faults may be inserted during a badly performed code editing. This problem is even more crucial for object-oriented languages (*e.g.* Java) where small changes can have major and nonlocal effects due to the use of subtyping and dynamic dispatch [26]. Binder [5] evidences this situation by listing a set of possible causes for a fault inclusion during a refactoring performance in OO codes: i) a codification mistake inserted by the programmer who was responsible for implementing the refactoring; ii) a non predicted interaction between the modified elements; and iii) a side effect introduced by an incorrect communication of elements of the system.

A number of attempts have been presented in the literature in order to avoid and/or handle the introduction of faults during refactoring edits (*e.g.* the following of micro steps combined with compilations/testing checking [10], the formal specification of refactoring edits [6], [18], the use of refactoring engines). But, they still have practical limitations due to either the inherited complexity and high costs to be applied or to the lack of precision to detect faults introduced by refactorings.

In order to detect refactoring faults, regression tests are often applied. A regression test suite [15] is composed of test cases which have already passed when run against previous versions of the SUT (System Under Testing) and should still pass when run against the next versions of it (*e.g.* system after refactoring edits). This practice has been extensively used in the industry [20]. However, regression testing activities are also very costly and time consuming. Studies (*e.g.* [15]) indicate that regression testing can consume almost half of the cost of software maintenance budget.

Due this fact, test case prioritization [25] has been used as a strategy for minimizing the costs related to regression testing execution and management. Test case prioritization has as goal reschedule the regression test cases in an order that favors the achievement of some test criterion (*e.g.* improve the rate of fault detection) more quickly. Thus, having a good prioritized test suite, the developers are able to run only the top test cases, or the amount of tests which is feasible due to resource limitations, without lose much of the testing potential. There are several test case prioritization techniques in the literature (*e.g.* [16], [33], [25]). They are usually based on coverage information and/or randomness. However, even though these techniques have been widely used producing good results in general, as they are general purpose techniques, when the goal is specifically to detect refactoring faults, they have shown to be unsuitable, producing bad and/or unstable results [2], [1].

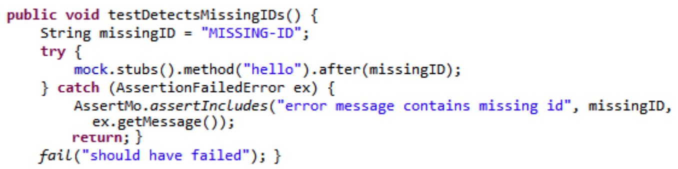
In this paper, we propose a refactoring-based selection/prioritization approach aiming at helping developers/testers to detect faults more effectively. The goal is to use information on how a refactoring edit can affect the SUT elements and how it can be related to the regression test cases. Firstly, we select the test cases which can be used for detecting a possible refactoring fault. Then, we propose an execution order of the whole test suite according to their chances of revealing refactoring faults. We evaluated our approach by performing a case study. From a real Java system, six fault versions were created, each one with a different refactoring fault. We compared our approach to other six widely known prioritization techniques, including one which focus on code



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

```
public MatchBuilder lookupID( String id ) {
    if (newMethod()) {
        throw new AssertionFailedError("no expected +
     invocation named '" + id + "'");      }
    return (MatchBuilder)idTable.get(id); }
 public boolean newMethod(id){
    return id == null;
 }
```

Fig. 1. Example of a badly performed refactoring in JMock's code.



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

Fig. 2. Failed Test Case.

changes. The results from this preliminary empirical study evidenced the promising results of our approach, where it outperformed the other techniques in most cases. Our approach is implemented for Java/JUnit systems in the PriorJ tool[1]. In summary, the main contributions of this paper are:

- An automated approach for selecting and prioritizing test cases aiming at improving the rate of refactoring fault detection (Section IV).
- The concept of refactoring fault models based on a given refactoring (Section V).
- A comparing evaluation using a real system and six well-known prioritization techniques (Section VI).

A motivation example and prelimary concepts are presented in Section II and III, respectively.

II. Motivational Example

In order to illustrate the problem addressed by this paper, consider *JMock*[2] – an open source library developed for helping the building of unit tests with Mock objects. The JMock project has about 5 KLOC, 504 JUnit test cases, and a test coverage of 92%. Suppose that, during code maintenance, a developer decides to simplify the condition expression of an IF command (*Decompose Conditional* edit [10]) presented in the *lookupID* method (class *org.jmock.Mock*) without changing its meaning, i.e. the developer wants to apply a refactoring. But, during the refactoring, the developer, by mistake, ended up changing the meaning of that condition. But, as no compilation error occurred, the developer did not noticed the fault and a subtle behavior change was added to the code. Figure 1 shows this example scenario.

By running the JMock regression suite, we can see that the only test case which reveals this fault is run after 334 others (the 335º test case), which is not a good position considering a scenario where the development team is under resource limitations. Even if we prioritize the suite by using coverage-based techniques such as Total and Additional Coverage along with Random Choice, the results may not be satisfactory. The best and the worst position where the failed test case is placed can vary from 61º (Total Statement Coverage) and 396º (Random Choice), respectively.

Looking closely at the failed test case (Figure 2), we can observe that it does not directly cover the method changed during the refactoring (*lookupID*). Situations like this can make fault debugging hard. The prioritization performed by the traditional coverage-based techniques do not help to solve this problem as they do not take into consideration any other aspect besides coverage data, producing test sequences where there is no semantic relation between the closest test cases.

III. Background

A. Program Refactoring

Code edits are a big part of the software development process. Those edits are usually classified as: i) *evolutionary*, changes related to adding or removing software functionalities; and ii) *refactorings*, changes related to structural modifications without any behavioral modification. The term *Refactoring* was coined by Opdyke [22], [21] and has been much disseminated since then, especially by Fowler [10]. Opdyke formally defined a refactoring change as a program transformation aiming at improving software quality aspects (reusability, maintainability, etc), but preserving its original semantics. The main idea of performing refactoring edits is to bring quality improvements to non-functional aspects of the software (*e.g.* clarity, readability, flexibility, performance).

Usually, the refactoring task is performed in an *ad hoc* way. Fowler [10] tried to reduce the chaos of performing a refactoring by establishing informal steps to be followed by the developers according to the type of edit intended. But, even carefully following these steps, most of the results of the edits end up being dependent on the developer's expertise, which can vary a lot. However, in the last few years refactoring tools (Eclipse IDE, Netbeans IDE) and techniques (*e.g.* [30]) have been developed in order to help developers to systematically perform refactoring edits, hopefully decreasing the chances of adding faults during this process. But, recent studies (*e.g.* [29]) have shown that even those widely used tools are not safe of fault inclusion.

B. Test Case Prioritization

As an alternative for decreasing the costs of regression testing and make re-testing more effective, *Test Case Prioritization* techniques [9] have been investigated with the goal of rescheduling test cases in an execution order that will satisfy a given test objective. The most common objective used for prioritization is to increase the rate of fault detection of a suite. Thus, usually, the prioritized suite will have in its top positions the test cases with the highest probability of revealing faults. The test case prioritization problem was formally defined by Rothermel [25] 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 $(\forall T'')(T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$

[1]http://sites.google.com/a/computacao.ufcg.edu.br/priorj/
[2]http://jmock.org/

where PT represents the set of possible orderings of T, and f is a function that gets the best possible values when applied to any ordering.

There are several approaches followed for performing test case prioritization (*e.g.* modification-based [34], fault-based [24], history-based [13]). But, in practice, the most used approach is the coverage-based prioritization [24], [25], [8], [31], [17]. The idea is that, as more coverage a test case achieve, more chances it has of revealing faults. In the sequel, we briefly define well-known prioritization techniques considered in this paper.

Total Statement Coverage (TSC): Schedules test cases based on the number of statements that each one covers.

Total Method Coverage (ASC): Similar to the TSC, differing by using the number of covered methods instead of covered statements as ordering criteria.

Additional Statement Coverage (ASC): The first test case is the one which covers more statements. The next test case to be chosen is the one which covers more statements that were not covered yet by the previously chosen test cases. This process is repeated until the prioritized test suite is complete.

Additional Method Coverage (AMC): Technique similar to the AMC, differing from ASC by using the coverage of methods as ordering criteria, instead of statement coverage.

Random (RD): Technique in which the test cases order in the prioritized test suite is defined randomly.

Srivastava and Thiagarajan (ST) [31]: This technique firstly identifies the changed blocks between the two versions (by binary analysis) and schedule the test cases according to how much of the changed block each test covers.

IV. Our Approach

This section gives a quick overview of our refactoring-based approach for selecting and prioritizing test cases. Figure 3 presents the general steps and artifacts produced, where: i) each rounded edge rectangle represents an activity; ii) each ellipse represents an artifact produced during an activity; and iii) an arrow indicates a flow between activities or an input/output of an artifact.

Initially, the following inputs are required: i) two versions of a SUT (the original and the one got after a refactoring edit); and ii) the SUT regression testing suite. As output, the approach produces: i) the set of test cases most related to the modification(s); and ii) a prioritized suite where the top tests are the ones more likely to reveal refactoring faults, according to our heuristics.

Briefly, our approach works in the following way. Firstly we have to determine, from the two versions of the program, the types of refactoring edits that were applied and where (activity *Discovering Refactoring Edits*). For that, we use the information from a refactoring plan document or from a tool support that automatically discovers it (e.g. Ref-Finder [14]). Being aware of what kind of refactorings were applied, we discover which code elements could be affected by those edits (activity *Discovering Impacted Elements*). For that, we proposed an analysis based on *Refactoring Fault Models*

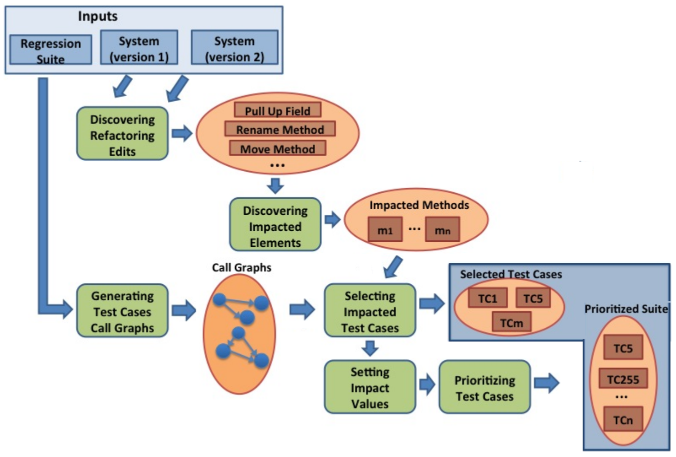


Fig. 3. Overview of our approach.

(Section V). Using these models, we extract from the source code the set of elements (methods, if we are dealing with Java systems) that can be negatively affected by a badly performed edit, we call it the "affected set" (AS).

At the same time, a Call Graph (CG) is extracted from each regression test case (activity *Generating Test Case Graphs*). These graphs show the SUT methods covered during the test case execution. They will be used for discovering which test cases are related to the AS elements. Then we perform a test selection to collect the test cases which cover the AS elements (activity *Selecting Impacted Test Cases*). Every test case whose call graph includes a call to at least one of the affected elements is selected during this activity. At the end the selected set is composed of test cases related to the edits. The output of this activity (the selection set) can be used by the developer/tester if prioritization is not needed.

In the sequence, in order to apply a prioritization, we associate an impact value with each test case (activity *Setting Impact Values*). This value corresponds to the amount of elements from AS that a test case covers. We associate an impact value 0 with the test cases that were not selected in the previous activity. Finally, prioritization is performed by using the impact values (activity *Prioritizing Test Cases*). The test case that covers the highest number of elements from AS (highest impact value) will be placed in the top; the second place of the prioritized suite will be occupied by the test case which covers the second highest number of elements from AS (second highest impact value); and so on. When two or more test cases have the same impact value, the decision of which one goes first will be randomly made. In other words, our prioritization heuristic is based on the assumption that a test case which covers more times elements from AS will be more likely to reveal a refactoring fault.

The *Refactoring Fault Models* (Section V) that we proposed are highly related to object-oriented languages characteristics (*e.g.* dynamic dispatch). Therefore, we believe that our approach originally can be used only in the OO context, even though the general ideas can be adjusted to other contexts.

V. Refactoring Fault Models

As refactoring faults, and how them may affect the system behavior, can vary a lot depending on the type of edit made, our approach is to focus test case selection/prioritizaton on the specific edit performed. For that, we proposed a set of *Refactoring Fault Models* (RFM). Those models were developed in order to relate each type of refactoring edit to the possible SUT elements that it can impact negatively in a way that affected code can be discovered by using testing coverage. The models were developed mainly based on the authors experience on refactoring and also on works in the literature which describe: i) proper ways to perform refactorings (*e.g.* [10]); and ii) formal pre and pos-conditions for guaranteeing that an edit is successfully made (*e.g.* [6]).

In order to obliterate any ambiguity and misunderstanding concerning to the RFM definition, we defined the *SJAS - Simplified Java Abstract Syntax* metamodel. SJAS is a simplified metamodel for the Java language. It was developed based on the *Abstract Syntax Tree Metamodel* [32] which is the one used by the Eclipse IDE for compilation checking. Considering a Java program as an instance of the SJAS metamodel, in addition to the informal representation of the RFM, we defined its formal representation by using ATL query rules [12]. These rules automatically return, by walking through the SJAS models, the set of possible affected elements related to a refactoring. Details about the definition of the whole RFM set (informal and formal definition) and about the SJAS metamodel can be see in this technical report [3]. We have already developed the RFMs for the six most common refactoring edits [19] (*Rename Method*, *Extracted Method*, *Move Method*, *Pull Up Field*, *Pull Up Method* and *Add Parameter*). Due the lack of space, in this paper, we informally introduce only the RFM of the refactoring *Rename Method*.

It is important to highlight that our idea by proposing the RFMs is not to substitute any deep impact analysis nor manual inspection of the source code. Our main goal was to identify the main problems that would come from a refactoring in a way that it could be easily connected to the regression test cases by using coverage information. Thus, our RFMs can be seen as a simplified testing-focused refactoring impact analysis.

A. Rename Method RFM

We can understand the Rename Method edit as a procedure *RenameMethod (C, m, n)* where *C* is the class where the method to be renamed is placed; *m* is the method which will be renamed; and *n* is the new name of method *m*. Then, we can informally define the *Rename Method RFM* as a set composed of the following methods:
- The *m* method in *C*.
- All methods in *C* that in its body there is a call to the *m* method.
- All methods in *C* that in its body there is a call to a method with the same signature of *n*.
- For each sub classes *S* of *C*:
  - All method in *S* that in its body there is a call to a method with the same signature of *m*.
  - All method in *S* that in its body there is a call to a method with the same signature of *n*.
- For each super class *Sp* of *C*:
  - All method in *Sp* which has the same signature of *m*.
  - All method in *Sp* which has the same signature of *n*.
- If *m* is a static method:
  - All method that in its body there is a call the *m*.

From this model, we can statically extract from a source code the methods that may be affected in a badly performed method renaming. The set of "possible impacted methods" will be used in the *Selecting Impacted Test Cases* and *Setting Impact Values* activities of our approach (Section IV).

VI. Case Study

In order to investigate on whether our approach is more suitable for discovering refactoring faults earlier than the other prioritization techniques presented in Section III-B, a case study was conducted. Besides the traditional coverage based techniques and the random one that are usually considered in empirical studies of test prioritization, we have also chosen the ST technique that is the closest one related to our approach, found in the literature, by focusing on using information about the changes performed.

A. Definition

The main goal of this case study is to *observe* on the *quantitative effectiveness of fault detection* when executing each ordering of test cases produced by the different techniques, for a given refactoring edit, from the point of view of the *testing execution team*. The main question followed is: What is the position of the first test case that fail in the ordering defined by each technique? The metrics chosen for answering this question are:
- **The** *F-Measure* **metric**. The number of distinct test cases to detect the first program failure;
- **The** *Average Percentage of Faults Detected* **(APFD) metric**. APFD values can be computed according to Equation VI-A. Higher APFD values imply faster fault detection rates.

$$APFD = 1 - \frac{TF_1 + TF_2 + ... + TF_m}{nm} + \frac{1}{2n} \qquad (1)$$

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 the fault i in the ordered test cases sequence.

Notice that, since we are considering only one refactoring edit and fault at a time, the APFD metric will mainly reflect a relative magnitude of the F-Measure value *w.r.t.* the size of the suite. However, this measure will be useful to analyze whether the different values of F-Measure are relatively comparable.

As empirical object we select a real Java open-source program (JMock[3]) described in Section II.

[3]http://jmock.org/

Based on the original code collected from the JMock repository, we manually created six faulty versions. In each of the versions, a fault was included related to a single badly performed refactoring edit (*Add Parameter*, *Extract Method*, *Move Method*, *Pull Up Field*, *Pull Up Method* or *Rename Method*). It is important to highlight that the person responsible for create those versions did that independently of the RFMs knowledge (Section V). Thus, we believe that the results produced in our study are not diverted. Table I-a shows the number of failed test cases in each of the refactoring version. We have to highlight two important aspects about the JMock faulty versions: i) the code of each of the versions vary from the original just by a refactoring edit; and ii) only one fault was included in each version and this fault was strictly related to the refactoring (details about the steps followed to introduce the fault can be seen in [3]). Each fault is introduced in isolation because: i) we wanted to see how our approach would behave in a situation where fewer test cases would fail, *i.e.* a situation that would be hard to perform selection and prioritization correctly; and ii) it is a hard task to automatically generate refactoring faults. Most of the empirical studies in regression testing use mutation for generating faults, but the kind of faults that those mutation operators create are not related to refactorings.

Using as input each of the versions (v1-v6) and its regression suite, we run a set of prioritization techniques and collected the respective prioritized suites. Tables I-b and I-c show the data collected in our case study, *F-Measure* and APFD values, respectively.

TABLE I
(A) NUMBER OF FAILED TEST CASES DUE A REFACTORING EDIT IN EACH VERSION; (B) POSITION OF THE FIRST FAILED TEST CASE; (C) APFD VALUES.

(a)

Version	Number of Failed Test Cases
v1 (Add Parameter)	2
v2 (Extract Method)	2
v3 (Move Method)	7
v4 (Pull Up Field)	6
v5 (Pull Up Method)	4
v6 (Rename Method)	3

(b)

Version	Position of the first failed test case (F-Measure)						
	Our Approach	ST	RD	TSC	TMC	ASC	AMC
v1	1	81	171.6	481	461	467	461
v2	1	1	137.2	191	231	123	219
v3	1	39	77.83	28	42	5	15
v4	6	41	63.3	3	2	61	55
v5	1	15	140	48	42	9	8
v6	5	1	105.6	340	327	299	304

(c)

Version	APFD						
	Our Approach	ST	RD	TSC	TMC	ASC	AMC
v1	0.999	0.840	0.660	0.047	0.086	0.074	0.086
v2	0.999	0.999	0.729	0.622	0.543	0.757	0.566
v3	0.999	0.924	0.846	0.945	0.918	0.991	0.971
v4	0.989	0.920	0.875	0.995	0.997	0.880	0.892
v5	0.999	0.971	0.723	0.906	0.918	0.983	0.985
v6	0.991	0.999	0.791	0.320	0.352	0.408	0.398

B. Analysis and Discussion

As we can observe from the collected data, our approach produced very good and stable results which give us preliminary evidence that it can be suitable for early discovering of refactoring faults in a practical context. By stable, we mean that the refactoring faults in each JMock faulty versions were, consistently, revealed after the execution of only very few test cases (in the worst case the fault was discovered in the 6º test case - v4). Also APFD values are all above 0.989, different from the results produced by the other techniques.

Comparing our technique to the other ones, we can see that our approach outperformed them in almost every case. There were just two cases where our prioritization put the failed test cases in a later position: i) *v4 - TSC/TMC*: in this case when we look at the first failed test case we can see that it covers a high number of code elements (statements and methods), i.e. it is a big test case. Though the early fault discovery, this prioritization may make the fault debugging harder, as a big number of elements to analyze can impair the discovery of the fault source; and ii) *v6 - TS*: in this case the heuristic of focusing the prioritization in the changed parts of the code produced a slightly better result than ours. Nevertheless, it is important to highlight that, even in the cases where the other techniques produced better results, our approach got very similar results w.r.t. the APFD metric (see the APFD results in Table I-b).

From our case study results we can also observe how stable our approach are when compared to other techniques. Observing the APFD values, we can see that our approach produced results varying in a very tight range [0.989; 0.999], *i.e.* we had a low standard variation ($\sigma = 0,004$) which do not occurred with the other techniques ($\sigma_{TS} = 0.06$; $\sigma_{RD} = 0.11$; $\sigma_{TSC} = 0.385$; $\sigma_{TMC} = 0.369$; $\sigma_{ASC} = 0.367$; $\sigma_{AMC} = 0.364$). Those facts corroborate the results that we got from previous empirical studies [2], [1] where we extensively tested those techniques (except from ST) for different refactoring faults and different regression suites. The results produced by those techniques in our studies were also very unstable. Thus, this study also gives evidence for the instability, and consequently the inadequacy, of the traditional techniques to find refactoring faults.

Another interesting aspect that we could observe from our case study was that our approach of using *Refactoring Fault Models* for selecting test cases in order to find refactoring faults seams to be precise. The reason is that in each version (*v1-v6*) all test cases that failed due to the refactoring were present in the set of selected test cases (output artifact of the *Selecting Impacted Test Cases* activity - Figure 3), i.e. the selection by using RFMs was powerful enough not to miss any important test case. As an example, in the selection performed in *v1* the returned set was composed by two test cases, the exactly two which failed due to fault related to the *Add Parameter* edit. We believe that having an efficient way to select refactoring related test cases, fault localization and debugging may be facilitated as the number of test cases to be executed and analyzed can be drastically reduced in a single step.

Finally, it is important to comment on threats to validity of the study and how they were handled to prevent undesired

effects on the observations obtained:
- *Incompleteness of the set of Refactoring edits.* The refactoring edits chosen are among the best known and widely applied.
- *Incompleteness of the RFMs*. Models were inspired by works presented in the literature which deals with formal conditions to check a refactoring edit;
- *Faults are not representative*. Faults were inserted following a refactoring practice.
- *JMock's suite do not represent the general JUnit suite*. The suite has a high test coverage (92%) and we only considered faults that could be revealed by the suite.

Therefore, because this is only a preliminary empirical study, focusing on a single application and a set of refactoring edits, and consequently with no statistical analysis, results cannot be generalize. However, the results present evidence that our approach can improve the practice of test case prioritization for refactoring edits.

VII. Related Work

Evidencing the important role that test case prioritization has played for decreasing the regression testing costs, there are many works proposing, developing and/or experimenting approaches/techniques in this sense. Yoo and Harman [35], for example, perform a survey showing the diversity of the prioritization approaches. In another recent work Singh at al. [27] summarize the state-of-art of the prioritization field by performing an elaborated literature review about prioritization. From an initial amount of 12,977 studies, they cataloged a great number of prioritization techniques which they classified among eight categories (*Coverage Based*, *Modification Based*, *Fault Based*, *Requirement Based*, *History Based*, *Genetic Based*, *Composite Approaches* and *Other Approaches*). Both works highlight the novelty that our approach brings as: i) none of the identified prioritization techniques are refactoring-based neither differentiate any refactoring aspect during the prioritization; and ii) the techniques which deal with code modifications do it by using model representations of the software behavior (*e.g.* finite state machines). Our approach is able to capture directly from the source code the possible problems related to refactoring edits, which, in most of cases, is more practical, and less costly.

Rothermel *et al.* [25] present several coverage-based techniques for prioritizing test cases and evaluate them according to their ability of improving the rate of fault detection. The authors opted for using code mutation as fault inclusion method and C programs as subjects of their studies. As result, the authors highlighted that even the last expensive prioritization technique was able to significantly improve the rate of fault detection of test suites. Additionally, Do *et al.* [7] performed a similar investigation where they showed that the traditional prioritization techniques are also effective on Java/JUnit systems. The conclusions of both works motivated our research: i) there is room for improvement of the traditional general purpose test case prioritization techniques; and ii) coverage data can also be useful for prioritization in OO systems.

Srivastava and Thiagarajan [31] propose a modification-driven prioritization technique which claims to be effective for early discovering faults added when edits are performed. The main idea of this technique is to identify the changed blocks between the two versions (by binary analysis) and schedule the test cases according to how much of the changed block each test covers. Although this technique deals with code changes, it does not differentiate the type of code changes applied (evolutionary or refactoring) neither apply any edit impact analysis for selection. These characteristics end up making this technique less efficient when we have a specific test objective for the prioritization (*e.g.* discover refactoring faults). Also, as no kind of change impact is analyzed, this technique showed to be not efficient when the fault added is related to a subtle side effect not related to the changed blocks.

About change impact analysis considering regressions artifacts, Ren *et al.* [23] propose an approach and a tool for analyzing the impact of changes in Java systems. First, they decompose the edits into atomic changes. Then, from the test cases call graphs and test execution traces they determine the subset of tests which is potentially affected by the changes. Finally, for each test case, they relate the atomic changes that may affect it. As this approach relates source code changes to test cases, it was really motivational for our work. We used a similar idea for correlate the possible affected code elements to the test cases by using call graphs analysis (but our call graph analysis is made differently). Though this work shows to be very promising, the authors evidences the need of improvement, as the set of "possible affected test cases" tends to be very big. As our approach for select/prioritize test cases is directly connected to how each refactoring edit was made, it tends to select a smaller and more accurate test case set. In [36] Zhang *et al.* extend Ren *et al.*'s work proposing an approach for ranking program edits in order to reduce the developers' effort on code inspection looking for faults. For that, they adapted spectrum-based fault localization techniques and use them to identify the edits more probable to introduce a fault. We envision this work as complementary to ours, as the adapted functions could also be used for refining our selection/prioritization process.

VIII. Concluding Remarks

This paper presents a new approach for selecting and prioritizing test cases using a refactoring-based strategy. The approach is centered on the use of *Refactoring Fault Models* which specialize test case analysis according to the type of refactoring made. After running a case study, we could observe that our approach produced, in general, better and more stable results when compared to the traditional prioritization techniques. Those facts give us evidence that the approach go towards to be effective for an early discovery of refactoring faults contributing to reduce the risks and costs of the refactoring activity, particularly in a software development environment with resource restrictions.

Based on the type of edit, the proposed approach produces: i) the set of test cases which are related to the edit and to its possible impacts; and ii) a prioritized suite where the test cases are scheduled according to its probability of discovering a refactoring fault. The PriorJ tool has an implementation of our approach which automatizes the whole process for Java/JUnit systems. As output, the PriorJ tool produces executable artifacts (JUnit test suites) that can be used directly during the refactoring application.

As further investigation we intend to: i) conduct a deeper evaluation based on a experiment with real programmers and a higher number of refactoring faults in order to measure the gains and drawbacks of using our approach; ii) investigate whether our refactoring-based selection/prioritization can improve fault debugging; iii) develop RFMs for more refactoring edit types; and iv) evaluate how fault discovering techniques (e.g. spectrum analysis) could be used in order to improve our prioritization process.

ACKNOWLEDGMENT

This work was supported by CNPq grants 484643/2011-8 and 560014/2010-4. Also, this work was partially supported by the National Institute of Science and Technology for Software Engineering[4], funded by CNPq/Brasil, grant 573964/2008-4. First author was also supported by CNPq.

REFERENCES

[1] E. L. G. Alves. Investigating test case prioritization techniques for refactoring activities validation: Evaluating suite characterists. Technical Report SPLab-2012-002, Software Practices Laboratory, UFCG, August 2012. http://splab.computacao.ufcg.edu.br/technical-reports/.

[2] E. L. G. Alves. Investigating test case prioritization techniques for refactoring activities validation: Evaluating the behavior. Technical Report SPLab-2012-001, Software Practices Laboratory, UFCG, May 2012. http://splab.computacao.ufcg.edu.br/technical-reports/.

[3] E. L. G. Alves. Test case prioritization based on refactoring fault models. Technical Report SPLab-2013-001, Software Practices Laboratory, UFCG, January 2013. http://splab.computacao.ufcg.edu.br/technical-reports/.

[4] K. Beck and C. Andres. *Extreme programming explained: embrace change*. Addison-Wesley Professional, 2004.

[5] R. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Professional, 2000.

[6] M.L. Cornélio. Refactorings as formal refinements. *Centro de Informática, UFPE. Doctoral Thesis*, 2004.

[7] H. Do, G. Rothermel, and A. Kinneer. Prioritizing junit test cases: An empirical assessment and cost-benefits analysis. *Empirical Software Engineering*, 11(1):33–70, 2006.

[8] S. Elbaum, A.G. Malishevsky, and G. Rothermel. *Prioritizing test cases for regression testing*, volume 25. ACM, 2000.

[9] S. Elbaum, A.G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *Software Engineering, IEEE Transactions on*, 28(2):159–182, 2002.

[10] M. Fowler and K. Beck. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[11] M.J. Harrold and A. Orso. Retesting software during development and maintenance. In *Frontiers of Software Maintenance, 2008. FoSM 2008.*, pages 99–108. IEEE, 2008.

[12] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez. Atl: a qvt-like transformation language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 719–720. ACM, 2006.

[13] J.M. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, pages 119–129. IEEE, 2002.

[14] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit. Ref-finder: a refactoring reconstruction tool based on logic query templates. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 371–372. ACM, 2010.

[15] H.K.N. Leung and L. White. Insights into regression testing [software testing]. In *Software Maintenance, 1989., Proceedings., Conference on*, pages 60–69. IEEE, 1989.

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

[17] C.L.B. Maia, R.A.F. do Carmo, F.G. de Freitas, G.A.L. de Campos, and J.T. de Souza. Automated test case prioritization with reactive grasp. *Advances in Software Engineering*, 2010, 2010.

[18] T. Mens and P. Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

[19] G.C. Murphy, M. Kersten, and L. Findlater. How are java software developers using the elipse ide? *Software, IEEE*, 23(4):76–83, 2006.

[20] A.K. Onoma, W.T. Tsai, M. Poonawala, and H. Suganuma. Regression testing in an industrial environment. *Communications of the ACM*, 41(5):81–86, 1998.

[21] W.F. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois, 1992.

[22] WF Opdyke and RE Johnson. Refactoring: An aid in designing object-oriented application frameworks. In *Proceedings of Symposium on Object-Oriented Programming Emphasizing Practical Applications*, 1990.

[23] X. Ren, F. Shah, F. Tip, B.G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of java programs. In *ACM SIGPLAN Notices*, volume 39, pages 432–448. ACM, 2004.

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

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

[26] B.G. Ryder and F. Tip. Change impact analysis for object-oriented programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 46–53. ACM, 2001.

[27] Y. Singh, A. Kaur, B. Suri, and S. Singhal. Systematic literature review on regression test prioritization techniques. *Special Issue: Advances in Network Systems Guest Editors: Andrzej Chojnacki*, page 379.

[28] G. Soares, B. Catao, C. Varjao, S. Aguiar, R. Gheyi, and T. Massoni. Analyzing refactorings on software repositories. In *Software Engineering (SBES), 2011 25th Brazilian Symposium on*, pages 164–173. IEEE, 2011.

[29] G. Soares, R. Gheyi, and T. Massoni. Automated behavioral testing of refactoring engines. *IEEE Transactions on Software Engineering*, 39(2):147–162, 2013.

[30] G. Soares, R. Gheyi, D. Serey, and T. Massoni. Making program refactoring safer. *Software, IEEE*, 27(4):52–57, 2010.

[31] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *ACM SIGSOFT Software Engineering Notes*, volume 27, pages 97–106. ACM, 2002.

[32] IBM TCS et al. Abstract syntax tree metamodel (astm). *OMG Document*, 2007.

[33] K.R. Walcott, M.L. Soffa, G.M. Kapfhammer, and R.S. Roos. Timeaware test suite prioritization. In *Proceedings of the 2006 international symposium on Software testing and analysis*, pages 1–12. ACM, 2006.

[34] W.E. Wong, J.R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *PROCEEDINGS The Eighth International Symposium On Software Reliability Engineering*, pages 264–274. IEEE, 1997.

[35] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

[36] L. Zhang, M. Kim, and S. Khurshid. Localizing failure-inducing program edits based on spectrum information. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 23–32. IEEE, 2011.

[4]www.ines.org.br