Regression Test Cases selection for Object-Oriented Programs based on Affected Statements

Samaila Musa, Abu Bakar Md Sultan, Abdul Azim Bin Abd-Ghani and Salmi Baharom
Faculty of Computer Science & Information Technology
Universiti Putra Malaysia, 43400 UPM serdang, Selangor, Malaysia
samailaagp@gmail.com

Abstract
One of the most important activities in software maintenance is Regression testing. The re-execution of all test cases during the regression testing is costly. And even though several of the code based proposed techniques address procedural programs, so many of them can’t be use directly on object-oriented programs. This paper presents modification-revealing test case selection for regression testing of object-oriented software using dependence graph model analysis of the source code. The experimental evaluation of our proposed approach was done using nine programs. We measured the performances of our selection approach using precision and inclusiveness metrics. It was observed from the results that our approach increase the efficiency and effectiveness of regression testing in term of precision and inclusiveness. It was concluded that selection of modification-revealing test cases based on affected statements provides considerably better results for precision and inclusiveness compared to retest-all and random technique, and reducing the cost of regression testing.

Keywords: Regression testing, test case selection, extended system dependence graph

1. Introduction
Software testing is a software engineering activity that never ends even after the software delivery. After delivering of software, if some changes are required in the software in term of improvement due to users requirements or debugging there is need to retest the software to make sure that existing functionalities and the initial requirement of the design are not affected, i.e., regression testing. But it is not feasible to retest-all test cases due to cost and time consumption especially when the test suite is big in size.

Regression testing is an activity that tests the modified program to ensure that modified parts behave as intended and the modification have not introduced sudden faults. The easiest way in regression testing is the tester simply executes all of the existing test cases to ensure that the new changes are harmless and is referred as retest-all method [1]. Retest-all is the safest technique, but it is possible only if the test suite is small in size. The test case can be selected at random to reduce the size of the test suite. But most of the test cases selected randomly can result in checking small parts of the modified software, or may not even have relation with the modified program. Regression test selection techniques will be an alternative approach. Regression test selection technique helps in selecting a subset of test cases from the test suite. The challenge in regression testing is identifying and selecting of best test cases from the existing test suite, and selecting good test cases will reduce execution time and maximize the coverage of fault detection.

The part of the system, either at the design level or codes level where modification has been done is tested for fault as it is more prone to errors. Regression test case selection helps in reduction of test cases which indeed helps in reduction of the testing cost. The
more safe and easy to make are the approaches that generate the model directly from the source code of the software.

Researchers have proposed many code-based approaches [2, 3, 4, 5] by identifying modifications in the level of source code, but the authors focus on the procedural-based programming which are not suitable in object-oriented programming widely today used in software development. Other researchers [1, 6, 7, 8, 9, 10, 11] address the issues of object-oriented programming but do not consider some basic concept of object-oriented features (such as inheritance, polymorphism, etc..) as bases in identifying changes.

A Control Call Graphs (CCG) based approach was presented [11] which is a reduced form of Control Flow Graph (CFG). This graph is a directed in which the nodes represent decision points, an instruction or a block of statements. An arc in the graph linking nodes Ni to Nj means that the statements corresponding to node Ni will be executed first, followed by the statements in node Nj. The control flow (method calls and distribution of the control flow) in the system are provided by CCG. The technique is more precise and captures the structure of calls and related control than the traditional Call Graph (CG). However, it is difficult to extracting information about changes in the source code that may not have direct impact on the method call.

A selection framework [12] was proposed that identify the affected nodes from extended system dependency graph, use the affected statements to select the test cases that are modification-revealing. But the framework was not implemented and evaluated.

In this paper a modification-revealing test case selection for regression testing is presented that selects test cases from existing test suite T used to test original program P by using Extended System Dependence Graph (ESDG) as an intermediate to identify the affected statements in the modified program P’ at statements level. MuJava mutation testing tool [13] is use to mutate the source code, represent testset in muJava in a JUnit test case form, run the mutants using muJava, and the tool will computes mutant score for the overall test cases, generate a report “mutant report” containing each mutant with the test cases that killed it. We use Fault-Exposing-Potential (FEP) metric [14] to perform mutation analysis in order to assess the quality of each test case, use the mutants to update the ESDG based on the changes, identify the affected statements from the ESDG using changed as slicing criterion to perform forward slicing, and select the affected test cases based on the affected statements as modification-revealing test cases. This approach will reduce the cost of regression testing by reducing the number of test cases to be used in testing the modified program.

The rest of this paper is organized as follows. The section that follows describes materials and methods. Section 3 presents the results. Section 4 presents the discussion. Section 5 presents the limitation of the study and Section 6 concludes the paper.

2. Materials and Methods

This section describes dependency graph, the evaluation matric (fault-exposing potential, precision and inclusiveness), regression testing and experimental design.

2.1. Dependence Graph

Extended System Dependency Graph (ESDG) based on the guides by [15] is used as intermediate to represent the programs. ESDG is used to model object-oriented programs, and can represents control and data dependencies, and information pertaining to various types of dependencies arising from object-relations such as association, inheritance and polymorphism. Analysis at statement levels with ESDG model helps in identifying changes at basic simple statement levels, simple method call statements and object-relations. ESDG is a directed, connected graph \( G = (V, E) \), that consist of set of V vertices and a set E of edges.
2.1.1. ESDG vertices: A vertex \( v \in V \) represents one of the four types of vertices, namely, statement vertices, entry vertices, parameter and polymorphic vertices.

2.1.2. ESDG edges: An edge \( e \in E \) represents one of the six edges, namely, control dependence edges, data dependence edges, parameter dependence edges, method call edges, summary edges, and class member edges.

2.2. Fault-Exposing Potential

Given program \( P \), test suite \( T \), and a set of mutants \( N = \{ n_1, n_2, n_m \} \) for \( P \), knowing which statement \( s_j \) in \( P \) contains each mutant. For each test case \( t_i \) in \( T \), execute each mutant version \( n_k \) of \( P \) on \( t_i \), knowing whether \( t_i \) kills that mutant using muJava. Having collected this information for every test case and mutant, consider each test case \( t_i \) and each statement \( s_j \) in \( P \), and the Fault-Exposing-Potential \( FEP(s, t) \) [14] of \( t_i \) on \( s_j \) as the ratio of mutants of \( s_j \) killed by \( t_i \) to total number mutants of \( s_j \):

\[
FEP(s, t) = \frac{\text{mutants of } s_j \text{ killed by } t_i}{\text{total number of mutants of } s_j}.
\]

2.3. Precision and Inclusiveness

2.3.1. Inclusiveness: Definition 2.3.1 [6]:

Suppose test suite \( T \) contains \( n \) test cases that are modification-revealing (detect fault) for original program \( P \) and modified program \( P' \), and suppose a technique \( M \) selects \( m \) (subset of \( n \)) of these tests. The inclusiveness of \( M \) relative to \( P, P' \), and \( T \) is:

The percentage given by the expression:

\[
\text{Inclusiveness} = (100(m/n)) \text{ if } n \neq 0
\]

Or Inclusiveness = 100% if \( n=0 \).

If for all \( P, P' \), and \( T \), \( M \) is 100% inclusive relative to \( P, P' \), and \( T \), \( M \) is safe.

2.3.2. Precision: Definition 2.3.2 [6]

Suppose test suite \( T \) contains \( n \) test cases that are non-modification-revealing (those that are not affected by the mutants) for original program \( P \) and modified program \( P' \), and suppose a technique \( M \) omits \( m \) (subset of \( M \)) of these tests. The precision of \( M \) relative to \( P, P' \), and \( T \) is:

the percentage given by the expression

\[
\text{Precision} = (100(m/n)) \text{ if } n \neq 0
\]

Or 100% if \( n = 0 \).

2.4. Regression Testing

This section presents our proposed approach for the selection of modification-revealing test cases \( T' \) from the test suite \( T \) to be used in testing the modified program \( P' \). Our proposed approach is based on the dependency graph representation of the source codes. The authors present regression test case selection framework (Figure 1) based on affected statements.
2.4.1. **Identify Changes**: The changes between the original program codes $P$ and the modified program $P'$ are identified in this stage by analysing the source code of the software. Use muJava [13] to mutate the source codes of the application. A file named “changed” is used to store the identified changes from class-level mutants and traditional-level mutants. The scopes of the changes in our approach are: Traditional level mutants and Class level mutants. The changed statements are used as slicing criterion to find all the affected statements by performing forward slicing on the updated model.

**Traditional level mutants**


**Class level mutants operators**

The class mutation operators are classified into four groups [17], based on the language features that are affected. Three of the groups are common to all object-oriented language features. The forth group is specific to Java programming object-oriented features. The four groups are: 1. Encapsulation 2. Inheritance 3. Polymorphism and 4. Java-Specific Features. Some examples of these features are shown in Table 1.
Table 1. Class Level Mutants Operators

<table>
<thead>
<tr>
<th>CATEGORY</th>
<th>OPERATOR</th>
<th>EXAMPLE</th>
<th>MUTANT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encapsulation</td>
<td>Access Modifier Change</td>
<td>public int s;</td>
<td>private int s; or</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>protected int s; or</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>int s;</td>
</tr>
<tr>
<td>Inheritance</td>
<td>Overriding method Deletion</td>
<td>class Car extends Vehicle {</td>
<td>class Car extends Vehicle {</td>
</tr>
<tr>
<td></td>
<td></td>
<td>void move(int a) {</td>
<td>//void move(int a) {</td>
</tr>
<tr>
<td></td>
<td></td>
<td>.......} }</td>
<td>.......}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>deleted method</td>
</tr>
<tr>
<td>Polymorphism</td>
<td>New method call with Child class type</td>
<td>Vehicle a; a = new Vehicle();</td>
<td>Vehicle a; a = new Car();</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>where Car is child class</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>of Vehicle</td>
</tr>
<tr>
<td>Java Specific features</td>
<td>Static Modifier Insertion</td>
<td>private boolean userAuthenticated;</td>
<td>private static boolean userAuthenticated;</td>
</tr>
</tbody>
</table>

An ESDG of a sample mutated source code is shown in Figure 2. In the source code, different mutants were introduced and represented in bold face, as shown in appendix A. The class ATM has 2 methods; this means class member edges are drawn from the class entry vertex Ce50 to the methods entry vertices (i.e., e51 and e61) as shown in Figure 2. PrePaidReload class also has 2 methods, class member edges are drawn from class entry vertex (Ce40) to the method entry vertices (e43 and e45) as shown in Figure 2. Call edges are drawn from the method call vertex to the callee method entry vertex; s59 and s70 (method call vertices) to e61 and e43 (method entry vertices) respectively. A method call edge is drawn from polymorphic method call vertex to the method entry vertex (e45). Control edges are drawn from statement vertex to the vertices that are control dependent on it; e51 to s53, s53 to s54, s54 to s55, s56, s57 and s58. Control edge from e61 to s62, s62 to s63, s65, s67, s69 and s71. The vertices s64, s66, s68 and s70 are control dependent on s63, s65, s67 and s69 respectively. The statement s64 is changed statement and the statement s71 is affected statement.

A class PrePaidRelod is added from statement ce42 to s45 that resulted in adding statements s58, s69 and the instantiated statement s70 as shown in appendix A. Adding statements s58, and s69 and s70 make s59 and s71 as changed statements respectively, which make statement s60 to be affected shown in Figure 2.
The changed statement s8 that is int c=2; being changed to static int c=2; where the word static is added making the variable c not been access directly as shown in appendix A. The statement s26 is data dependent on the changed statement s8, therefore s26 is affected statement. The statement e21 is a deleted constructor for class Balance that makes the instantiated statements of balance to used default constructor of class Balance. Identify all the statement that instantiate the object balance as changed statements and statements that dependent on the instantiated object of balance type are affected statements. Vertex s64 is changed statement since its instantiate the object balance and vertex s71 depends on s64, therefore it is marked as affected statement as shown in Figure 3.

The statement s29 which is a method overloading call replaced the body of another overloaded method, this makes the overloaded method e28 to be changed statement and all method calls to this overloaded method are affected statements. Statement s37 is a deletion of super call statement that makes reference to user parent constructor declaration. The deletion of super statement makes reference to default constructor of the parent class Transaction. The statement e36 is changed statement and statement s68 as affected statement as shown in Figure 3. The statement s40 is changed from Withdrawal.getX () to Withdrawal.getY () and is marked as changed statement. Statements that dependent on s40 are the affected statements.
2.4.2. Test case selection: This approach use algorithm I to select test cases based on the affected statements. The affected statements are identified from the updated model M'.

If the following test cases have the following coverage information:

- $t_1 = \{n1 \ n2 \ n3 \ n4\}$
- $t_2 = \{n1 \ n2 \ n3 \ n6 \ n7 \ n9\}$
- $t_3 = \{n1 \ n2 \ n4 \ n5 \ n6 \ n10\}$
- $t_4 = \{n1 \ n12\}$
- $t_5 = \{n1 \ n9 \ n10\}$
- $t_6 = \{n1 \ n2 \ n3 \ n4 \ n9 \ n10 \ n13\}$
- $t_7 = \{n10 \ n13\}$
- $t_8 = \{n1 \ n2 \ n3 \ n6\}$
- $t_9 = \{n1\}$
- $t_{10} = \{n1 \ n11\}$
Assuming the affected nodes/statements are n2 n3 n4 n6 n7 n9 n10 and n13, generated by lines 11 to 14 of algorithm 1 and then based on lines 15 to 20, the selected test cases will be:

\[ T' = \{ t1, t2, t3, t5, t6, t7, t8 \} \]

and their affected nodes will also be:

- \( t1 = \{ n2, n3, n4 \} \)
- \( t2 = \{ n2, n3, n6, n7, n9 \} \)
- \( t3 = \{ n2, n4, n6, n10 \} \)
- \( t5 = \{ n9, n10 \} \)
- \( t6 = \{ n2, n3, n4, n9, n10, n13 \} \)
- \( t7 = \{ n10, n13 \} \)
- \( t8 = \{ n2, n3, n6 \} \)

This means that all the test cases that are affected will be selected for regression testing. If there is addition of new functionality to the system, the new test cases are added to the selected test cases.

**ALGORITHM 1** Algorithm for selecting test cases

```java
1 EvolRegTCaseSel (M', T, changed, affectedNodes, T') {
2     changed: is the set of changed objects
3     M': is the updated extended system dependence graph
4     tc: is a test case
5     T': is the set of test cases from the newly added functionalities
6     T = \{ tc | tc ∈ T\}
7     nDep: are the set of nodes that depend on changed nodes
8     affectedNodes: is the set of changed and nDep nodes
9     T': is the set of selected test cases based on the affectedNodes
10    affectedNodes = \{ \}
11    T' = \{ \}
12    For (node n : Changed) {
13        Find nodes (nDep) that are dependent on node n
14        affectedNodes = affectedNodes ∪ nDep
15    }
16    While (affectedNodes! = Null) DO {
17        For (node a : affectedNodes) {
18            Find all test cases (tc) that cover node a
19            T' = T' ∪ tc
20        }
21    }
22    If new functionalities are added with the test cases T"
23    T' = T' ∪ T"
24    return T'
25 }
```

2.5. Experimental Design

Experiment are conducted to evaluate the quality of the test case generation in term of fault-exposing potential (FEP), and also evaluate the precision and inclusiveness of our selection technique (PA), random method (RM) and retest-all (RA). We run the experiment on the same computer using JAVA jdk1.7 with Neat beans IDE on Intel ® Core™ i5-3470 at 3.2GHz and 8 GB RAM, under Microsoft Window 7 Professional.
2.5.1. Goal of the Study: The goals of our study are to evaluate the precision and inclusiveness of our selection approach based on affected statements. This research will be important to software testers in selecting test cases for regression testing. Our research questions are:

RQ1. How effective is the selection technique precision based on affected statements?
RQ2. How effective is the selection technique inclusiveness based on affected statements?

To address the questions above, the authors carry out experiments to evaluate the performances based on precision and inclusiveness. Based on the above research questions, the following hypotheses are presented:

- The null hypothesis 1 (H0 prec) = There is no significant difference in the mean of precision of using our proposed selection technique based on affected statements P(A), random selection (RM) and retest-all (RA), and can be formulated as:
  \[ H_{0\text{prec}} = \mu_{PA} = \mu_{RM} = \mu_{RA} \]
  Where \( \mu_j \) is the mean rate of precision scores of technique j in using selection technique measured on the nine programs.

- Alternative hypothesis 1 (H1 prec) = There is significant difference in the mean of precision of our proposed approach in using affected statements to select test cases P(A), random selection (RM) and retest-all (RA). It can be formulated as:
  \[ H_{1\text{prec}} = \mu_{PA} \neq \mu_{RM} \neq \mu_{RA} \] (one mean is different from the others)

- The null hypothesis 2 (H0 inclu) = There is no significant difference in the mean of inclusiveness of using our proposed approach selection technique based on affected statements P(A), random selection (RM) and retest-all (RA), and can be formulated as:
  \[ H_{0\text{inclus}} = \mu_{PA} = \mu_{RM} = \mu_{RA} \]
  Where \( \mu_j \) is the mean rate of inclusiveness scores of technique j in using selection technique measured on the nine programs.

- Alternative hypothesis 2 (H1 inclu) = There is significant difference in the mean of inclusiveness of using our proposed approach in using affected statements to select test cases P(A), random selection (RM) and retest-all (RA). It can be formulated as:
  \[ H_{1\text{inclus}} = \mu_{RA} \neq \mu_{PA} \neq \mu_{PA} \] (one mean is different from the others)

2.5.2. Experimental Setup: The empirical procedure for our proposed approach is: given a source code for each program, i.e., CC (cruise control) [18], BS (Binary Search Tree) [19] and AT (ATM case study) [20], construct extended system dependency graph for each program, use graphviz to represent the graph, use muJava to mutate the programs, perform mutation testing by building junit into the muJava by representing testset in muJava in aJUnit test case form in order to have mutants killed by each test case, run the mutants using muJava; the tool computes mutant score for the overall test cases and generate a report “mutant report” containing each mutant with the test cases that killed it, use Fault-Exposing-Potential (FEP) to perform mutation analysis for each test case to assess it quality using the mutant report, reflect the changes in the extended system dependence graphs to have updated graph, get all the affected statements/nodes that are dependent on the changes, use Path-Based Integration testing to generate and save the coverage information for each test case t1, t2,…,tn, select which test cases in T are modification-revealing with respect to the affected statements, and compute the precision and exclusiveness.

Three programs with three different versions each are used for empirical evaluation of our proposed approach which make up nine (9) programs; the cruise control (CC) is from a Software-artifact Infrastructure Repository (SIR); a repository that provide software for experimentations, binary search tree (BS) is from sanfoundry technology education blog, and ATM machine (AT) is a case study provided in Java how to Program book for the implementation of the ATM machine, as shown in Table 2; SPr is the sample program, LOC is the lines of code, NC is the number of classes for each version, NM is the number of methods, NT is the number of test cases, NMT is the number of mutants, CC1, CC2 and
CC3 are the Cruise control program versions 1, 2 and 3 respectively, BS1, BS2 and BS3 are binary search tree program versions 1, 2 and 3 respectively, and AT1, AT2 and AT3 are the ATM program versions 1, 2 and 3 respectively.

At CC1, brake() method and part of handleCommand() method are mutated to have 36 mutants. At CC2 methods accelerate(), engineOff(), and part of handleCommand() are mutated in addition to methods mutated in CC1 to have 84 mutants. At CC3 the methods engineOn(), enableControl(), disableControl(), part of handleCommand() are mutated, and the constructor CarSimulator() deleted together with the mutants from version 2 to have 167 mutants. At BS1, the methods countNode(), and part of delete(), and also some class-level mutants are mutated to have 52 mutants. At BS2, the methods isEmpty(), and part of delete() are mutated to have 43 mutants. At BS3 the methods insert() and search() are mutated and also some class-level mutants to have 119 mutants. At AT1, the method execute() of deposit and balanceInquiry together with their constructors and the method credit() of bankDatabase and Account classes are mutated to have 49 mutants. At AT2 in addition to mutants in version 1, methods isSufficientCash() and dispenseCash() of class CashDispenser with its constructor, debit() of class Account, and part of the method execute() of class Withdrawal are mutated and some class-level mutants to have 120 mutants. At AT3 the method validatePin() of class Account with its constructor, the method authenticateUser() of class BankDatabase with it constructor and part of the method execute() of class Withdrawal are mutated, and some class-level mutants together with mutants in version 2 to have 191 mutants.

<table>
<thead>
<tr>
<th>SP</th>
<th>LOC</th>
<th>NC</th>
<th>NM</th>
<th>NT</th>
<th>NMt</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC1</td>
<td>283</td>
<td>4</td>
<td>33</td>
<td>10</td>
<td>36</td>
</tr>
<tr>
<td>CC2</td>
<td>310</td>
<td>5</td>
<td>36</td>
<td>13</td>
<td>84</td>
</tr>
<tr>
<td>CC3</td>
<td>339</td>
<td>6</td>
<td>38</td>
<td>15</td>
<td>167</td>
</tr>
<tr>
<td>BS1</td>
<td>293</td>
<td>3</td>
<td>25</td>
<td>17</td>
<td>52</td>
</tr>
<tr>
<td>BS2</td>
<td>323</td>
<td>4</td>
<td>28</td>
<td>20</td>
<td>43</td>
</tr>
<tr>
<td>BS3</td>
<td>343</td>
<td>5</td>
<td>30</td>
<td>22</td>
<td>119</td>
</tr>
<tr>
<td>AT1</td>
<td>670</td>
<td>12</td>
<td>42</td>
<td>23</td>
<td>49</td>
</tr>
<tr>
<td>AT2</td>
<td>740</td>
<td>13</td>
<td>47</td>
<td>26</td>
<td>120</td>
</tr>
<tr>
<td>AT3</td>
<td>770</td>
<td>14</td>
<td>50</td>
<td>28</td>
<td>191</td>
</tr>
</tbody>
</table>

3. Results

The data collected from the experiments are shown in Table 3. Table 3 gives the results of the nine (9) programs, Where SP is the sample program, NT is the number test suite, Mr is the number of test cases that are modification revealing in NT; a test case is said to be modification-revealing if it causes the result of the original and mutated programs to be different computed using the procedure stated in section 2.2, #Tc is the percentage of NT of selected test cases; #Tc is the percentage of test cases selected by a technique, Precision (Prec) and Inclusiveness (Incl) as metrics for assessing the results. Precision measures the percentage of non-modification-revealing tests omitted in the selected tests, whereas inclusiveness is used to measure the percentage of modification-revealing tests selected by the regression testing technique as described in Section 2.3. The three columns of the results in percentages represent our proposed approach, random selection and retest-all methods.

We measure the quality of the test cases generated by computing the fault-exposing potential of each test case on each version, and the results are shown in Figures 4, 5 and 6.
Figure 4 shows the FEP of CC1, CC2 and CC3; indicating how best a test case is in term of fault coverage, and the modification-revealing test cases for each version. CC1 has only two modification-revealing test cases (T5 and T6), so any selection technique that selects any of test case T1, T2, T3, T4, T7, T8, T9 or T10 is not 100% precise, and any selection technique that does not select T5 or T6 is not 100% inclusive. In CC2 and CC3, the modification-revealing test cases are T2, T3, T4, T5 and T6, and T1, T2, T3, T4, T5, T6, T7, T8, T9 and T10 respectively.

Figure 5 shows the FEP of BS1, BS2 and BS3; indicating how best a test case is in term of fault coverage and the modification-revealing test cases for each version. CC1 has only four modification-revealing test cases i.e., T10, T11, T14 and T15, so any selection technique that selects any of test case T1, T2, T3, T4, T5, T6, T7, T8, T9, T12, T13, T16 or T17 is not 100% precise, and any selection technique that does not select any of T10, T11, T14 or T15 is not 100% inclusive. In BS2 and BS3, the modification-revealing test cases are T4, T5, T6, T7, T8, T9, T10, T11 and T16, and T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15, T16 and T17 respectively.
Figure 6 shows the FEP of AT1, AT2 and AT3; indicating how best a test case is in term of fault coverage and the modification-revealing test cases for each version. AT1 has only four modification-revealing test cases i.e., T3, T21, T22 and T23, so any selection technique that selects any of test case T1, T2, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15, T16, T17, T18, T19 or T20 is not 100% precise, and any selection technique that does not select any of T3, T21, T22 or T23 is not 100% inclusive. In AT2 and AT3, the modification-revealing test cases are T3, T6, T7, T10, T11, T13, T14, T16, T17, T19, T20, T21, T22 and T23, and T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, T11, T12, T13, T14, T15, T16, T17, T18, T19, T20, T21, T22 and T23 respectively.

![Figure 6. Fault-Exposing Potential of the Three Versions of ATM](image)

Table 3 shows the advantages of our approach in term of precision and inclusiveness over the two methods. From the table, only 20% of the test cases are needed to have 100% inclusiveness and precision by our proposed approach in CC1. The CC2 and CC3 programs require 23% and 67% respectively to obtain 100%, whereas RM with the same number of test cases has precision of 13%, 30%, and 80% and inclusiveness of 50%, 0% and 60% for the three versions of CC. Retest-all needs 100% of the test cases in all the three versions to have precision of 0% and 100% inclusiveness for each version. The three versions for binary search tree BS1, BS2 and BS3 require 24%, 40%, and 77% respectively to achieve precision of 100% and inclusiveness of 100% each by our proposed approach, while the random selection achieved precision of 23%, 42% and 60%, and inclusiveness of 25%, 38% and 82% respectively with the same number of selected test cases with our approach. Retest-all requires 100% of the test cases in all the three versions to achieve precision of 0% and inclusiveness of 100% for each version. For AT1, AT2 and AT3, our approach requires 17%, 19% and 82% to achieve 100% precision and 100% inclusiveness for each version respectively, whereas with same number of test cases, random selection achieve 1%, 38% and 40% precision and 50%, 40% and 91% inclusiveness for each version respectively. Retest-all requires 100% of the test cases in all the three versions to achieve precision of 0% and inclusiveness of 100% for each version.
Table 3. Results of Three Programs with 3 versions each

<table>
<thead>
<tr>
<th>SP</th>
<th>NT</th>
<th>Mr</th>
<th>Proposed technique (PA)</th>
<th>Randomly select (RM)</th>
<th>Retest-all (RA)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#Tc</td>
<td>Prec</td>
<td>Incl</td>
<td>#Tc</td>
<td>Prec</td>
</tr>
<tr>
<td>CC1</td>
<td>10</td>
<td>2</td>
<td>20</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>CC2</td>
<td>13</td>
<td>5</td>
<td>38</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>CC3</td>
<td>15</td>
<td>10</td>
<td>67</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>BS1</td>
<td>17</td>
<td>4</td>
<td>24</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>BS2</td>
<td>20</td>
<td>9</td>
<td>50</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>BS3</td>
<td>22</td>
<td>17</td>
<td>77</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>AT1</td>
<td>23</td>
<td>4</td>
<td>17</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>AT2</td>
<td>26</td>
<td>14</td>
<td>54</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>AT3</td>
<td>28</td>
<td>23</td>
<td>82</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

3.1. Analysis and Comparison of Precision

The results of 100% precision in the entire subject program indicate that our proposed approach (PA) performs well on omitting non-modification-revealing test cases from the subset selected for regression testing, as shown in Figure 7. While retest-all (RA) performed poorly in term of precision, since all the non-modification-revealing test cases are selected. The random method omits test case blindly without considering the modifications.

![Figure 7. Percentages of Non-Modification-Revealing Omitted by each Approach](image)

3.1.1. Hypothesis testing (H\textsubscript{0pro}) To determine the rejection or acceptance of null hypothesis, R statistical tool was used to conduct the test. The parametric ANOVA test was used with 5% significance level. The authors obtained a F-value of 110.24 which was greater than the critical value of 3.4028 (F\textsubscript{crit} = F\textsubscript{0.05(2, 24)}) for the F-distribution at the degree of freedom of 2 and 24 and 95% confidence interval for the difference among treatments (approaches). Based on the decision rule, H\textsubscript{0} is rejected if F-value > F\textsubscript{crit} (110.24 > 3.4028) or if p-value < α (8.013e-13 < 0.05). With the respect of the above data, Ho is rejected. From the analysis, it shows that there is a significant difference in mean rate of fault detection for the three approaches.

Since there is significant difference between the approaches in term of precision, and ANOVA test cannot tell which specific approach was significantly different from the
others, proceed to testing the main effect pairwise comparisons. To accomplish this, apply pairwise.t.test() function to our independent variable (selection approach), and the result is shown in Table 4. The table shows that each approach is compared with the two other approaches. From the table, the approach PA compared to the approaches RA and RM revealed very strong significant differences. Approach RM compared to the RA revealed a very strong significant difference. This means that selection approach PA have overwhelming evidences to support the alternative hypothesis.

**Table 4. Analysis of Variance Table for Precision**

<table>
<thead>
<tr>
<th></th>
<th>PA</th>
<th>RA</th>
</tr>
</thead>
<tbody>
<tr>
<td>RA</td>
<td>1.7e-13</td>
<td>-</td>
</tr>
<tr>
<td>RM</td>
<td>9.0e-06</td>
<td>3.0e-09</td>
</tr>
</tbody>
</table>

From the analysis above, PA and RM are significantly better than RA. This analysis provided the evidence that a pair of treatment mean is not equal. Finally, it shows that PA might be more precise compared to RA and RM.

### 3.2. Analysis and Comparison of Inclusiveness

The results of 100% inclusiveness in the entire subject programs indicate that our proposed approach performs well on selecting modification-revealing test cases from the subset selected for regression testing, while retest-all required 100% of the test cases to achieve the same performance with our approach that select only part of the test cases, as shown in Figure 8. So also randomly selection method performs poorly in term of inclusiveness in most of the case studies, since it does not consider modification made.

![Inclusiveness](image.png)

**Figure 8. Percentages of Modification-revealing Selected by each Approach**

#### 3.2.1. Hypothesis testing ($H_{0\text{inclus}}$): The F-value 30.814 which was greater than the critical value of 3.4028 ($F_{\text{crit}} = F_{0.05(2, 24)}$) for the F-distribution at the degree of freedom of 2 and 24 and 95% confidence interval for the difference among treatments (approaches). Based on the decision rule, $H_0$ is rejected since F-value > $F_{\text{crit}}$ (30.814 > 3.4028) and P-value < $\alpha$ (2.351e-07 < 0.05). From the analysis, it shows that there is a significant difference in mean of inclusiveness for the three selection approaches. Since there is significant difference between the approaches, proceed to testing the main effect pairwise comparisons using pairwise.t.test() function and the result is shown in Table 5. The table
shows that each approach is compared with the two other approaches. From the table, the approach PA compared to the approaches RA and RM revealed no significant and very strong significant differences respectively. Approach RA compared to RM revealed a very strong significant difference, but RA used 100% of the test cases compared to PA and RM that used some parts of them. This means that there is overwhelming evidence to support the alternative hypothesis.

<table>
<thead>
<tr>
<th>Table 5. Analysis of Variance Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>PA</td>
</tr>
<tr>
<td>----</td>
</tr>
<tr>
<td>RA</td>
</tr>
<tr>
<td>RM</td>
</tr>
</tbody>
</table>

This analysis provides the evidence that a pair of treatment mean is not equal. Finally, it shows that PA might have a better performance in term of inclusiveness compared to RA and RM.

4. Discussion

What were observed and analysed in the preceding section are discussed in this section. The results of our empirical study of the three selection techniques (PA, RM and RA) drawn from the nine programs show that PA is able to provide better results than the other two approaches. Our approach needs only parts of the test cases to achieve better results.

In term of efficiency, PA requires less time to detect all the faults when compared with retest-all. For example, PA requires 20%, 24% and 17% of the test suite to detect all the faults in CC1, BS1 and AT1 respectively. This means at CC1, there is only need of 1/5 of the time required to detect all the faults, 1/4 of the testing time of BS1 to detect all the faults, and 1/5 of the testing time of AT1 to detect all the faults when compare with retest-all that requires 100% of the test time to detect all the faults, and random technique (RM) that requires the same time with PA but does not detect all the faults.

PA requires 38%, 50% and 54% of the test suite to detect all the faults in CC2, BS2 and AT2 respectively. This means at CC2, there is only need of 2/5 of the time required to detect all the faults, 1/2 of the testing time of BS2 to detect all the faults, and 3/5 of the testing time of AT2 to detect all the faults when compare with retest-all that requires 100% of the test time to detect all the faults, and random technique (RM) that requires the same time with PA but does not detect all the faults.

PA requires 67%, 77% and 82% of the test suite to detect all the faults in CC3, BS3 and AT3 respectively. This means at CC3, there is only need of 2/3 of the time required to detect all the faults, 4/5 of the testing time of BS3 to detect all the faults, and 4/5 of the testing time of AT3 to detect all the faults when compare with retest-all that requires 100% of the test time to detect all the faults, and random technique (RM) that requires the same time with PA but does not detect all the faults.

Based on the above results, the selection technique is less efficient when the entire code is affected as shown in versions 3 of the sample programs (CC3, BS3 and AT3). But the selection technique performs better in CC1, BS1 and AT1, so also in CC2, BS2 and AT2 where the changes affect only some part of the code.

In term of precision, PA performs better in all the nine sample programs; this means that RA does not select any non-modification-revealing test cases from the test suite. This may be due selection based on affected statements generated from ESDG of the source code. RM performs poorly in term of precision, because it selects test cases without considering the modification, as such it will select as many as it can the non-modification-revealing test cases, while RA selects all the non-modification-revealing test case because it select all the test cases whether modification or non-modification-revealing, and resulted in performing very poorly; zero scores in all the sample programs.
In term of inclusiveness, our approach (PA) and retest-all (RA) perform better in all the nine sample programs; this indicates that the two techniques select all the modification-revealing test cases but PA does not requires 100% of the test cases to achieved the same results with RA that requires 100%, while RM performs poorly compared with the PA and RA. This may be due the selection of the test cases at random without considering the modifications. Our approach (PA) used the same percentages of selected test cases with random technique (RM), but generally PA performed better than the RM, this may be due to the fact that RM does not consider any criteria for selecting the test cases.

The results of the above experiment show that there was statistically significant difference between the selection techniques based on affected statements than selecting all the test cases or randomly select test cases. From the results, our approach requires less number of test cases compared to retest-all, and with the same number of test cases with random selection; our approach achieved higher percentages in precision and inclusiveness. This means the proposed approach does reduce the time and effort required for regression test case selection after program modification, compared with retest-all that blindly selects all test cases and random selection that select test cases without considering the modification made.

5. Limitations of the Study

Our proposed approach is based on the codes of the software, which might be time consuming when applied to software that has very large number of LOC (line of codes) which becomes its limitation. Also, if the test cases are very small in size, there would be no need of selection as retest-all will be feasible and selection time might be greater than the execution time of retest-all. We also note that our proposed approach assumes that the ESDG are updated in a timely way, every time changes are introduced into the programs. This assumption also becomes a limitation of our approach.

6. Conclusions

A regression test case selection approach that selects test cases $T^*$ based on affected statements was proposed, use muJava to mutate the original program, used extended system dependence graph to represents the source code of the original program and update the graph whenever the program is modified, store the changes in a file named changed, and generate coverage information for each test case from the source code using Path-Based Integration testing. The changed information is used to identify the affected statements from the updated model using changed as slicing criterion to perform forward slicing on the ESDG, and select test cases that were identified based on the affected statements.

The effectiveness of the approach was evaluated using precision and inclusiveness. In this paper, three sample programs with three different versions each are used for the evaluation. From the results presented, PA provides better results in the nine programs in term of precision, and PA and RA perform the same in term of inclusiveness but our approach (PA) needs only some parts of the test cases while RA required 100% of the test cases to perform the same. PA select test cases more effectively compared to using RA and RM which result in reducing the cost of regression testing.

As a feature work, we will present an approach that will select and prioritize the selected test cases in order to improve the efficiency and effectiveness of test cases for regression testing.
Appendix A

c.1. Public class Transaction {
   s.2. public Transaction (int userAccountNumber, Screen atmScreen, BankDatabase atmBankDatabase) {
      s.3. accountNumber = userAccountNumber; s.4. screen = atmScreen;
      s.5. bankDatabase = atmBankDatabase;
   } // end constructor transaction
   s.6. abstract public void execute( );
} // end class transaction
e.7. Class Withdrawal extends Transaction {
   s.8. static int c=2; // static added
   s.9. Public int d; // initial value deleted
   s.10. public Withdrawal () {
      s.11. super();
      s.12. }
   s.13. public void execute() { ........... }
   s.14. public int getX() { s.15. return x; }
   s.16. public int getY() { s.17. return y; }
   s.18. ......................... } // end class withdrawal
e.20. Class Balance extends Transaction {
   e.21. // public Balance( ) {
   e.22. super(); } // constructor deleted
   e.23. public void execute() { ............. }
   e.24. void Add (int a) {
   e.25. this.Add(a); //replacing the body
   }
} // end class balance
e.35. Class Deposit extends Transaction {
   e.36. public Deposit( ) {
   e.37. } // super(); } // deleted super call
   e.38. public void execute() {
   e.39. Withdrawal.getX( ); // Instate of getX( )
   e.40. }
   e.41. Balance.Add(x, y);
} //end method execute

e.42. Class PrePaidReload extends Transaction{ // added class
   e.43. public PrePaidReload( ) {
   e.44. super();
   e.45. public void execute() { ............. }
} // end class prepaidreload
   //user chose to perform
   s.55. case BALANCE_INQUIRY:
   s.56. case WITHDRAWAL:
   s.57. case DEPOSIT:
   s.58. case PrePaidReload:
   //INITIALIZE AS NEW OBJECT
   s.59. currentTransaction = createTransaction(mainMenuSelection);
   s.60. currentTransaction.execute();
   s.61. break;
   s.62. } //end switch
} //end method performTransaction

e.61. private Transaction createTransaction(int type) { //return object of specified transaction
   e.62. switch (type) //determine which type of transaction to create
   e.63. {
   e.64. case BALANCE_INQUIRY:
   e.65. temp = new Balance(currentAccntNu, screen, bankDbase);
   e.66. case WITHDRAWAL:
   e.67. temp = new Withdrawal(currentAccntNu, screen, bankDbase, keypad, cashDispenser);
   e.68. case DEPOSIT:
   e.69. temp = new Deposit(currentAccntNu, screen, bankDbase, keypad, depositSlot);
   e.70. case PrePaidReload:
   e.71. temp = new PrePaidReload(currentAccntNu, screen, bankDbase, keypad, depositSlot);
   e.72. } //end switch 
   e.73. return temp;
} // end method createtransaction
} // end class ATM
References