A Program Model Based Regression Test Selection Technique for Object-Oriented Programs

Nitesh Chouhan
Assistant Professor
Department of IT
MLVTEC Bhilwara

Dr. Maitreyee Dutta
Associate Professor
Department of Computer Science
NITTTR, Chandigarh

Dr. Mayank Singh
Associate Professor
Department of Computer Science
KEC, Ghaziabad

***Abstract*- We propose a regression test selection technique that is based on analysis of source code of an object-oriented program. First we construct a System dependency graph model of the original program from the source code. When some modification is executed in a program, the constructed model is updated to reflect the changes. Our approach in addition to capturing control and data dependencies represents the dependencies arising from object-relations. The test cases that exercise the affected model elements in the program model are selected for regression testing. In our approach System Design Graph representation will be used for regression test selection for analyzing and comparing the code changes of original and modified program. Empirical studies carried out by us show that our technique selects on an average of 26.36. % more fault-revealing test cases compared to a Control Dependence Graph based technique while incurring about 37.34% increase in regression test suite size.**

Keywords: Software maintenance, Regression testing, Regression test selection, System Dependence Graph.

I. INTRODUCTION

Maintenance of an object oriented program is frequently necessitated to fix bugs, to enhance or adapt existing functionalities. Figure1, adapted from Do et al. [7], shows a popularly-followed maintenance process model. After that resolution tests are carried out to verify the modified parts of the code, while regression testing is carried out to test the unchanged parts of the code that may be affected by the code change. After the testing is complete, the new version of the software is released, which then undergoes a similar cycle. In the development phase, regression testing may begin after the detection and correction of errors in a program. At the last stages of program development when the program has been reasonably tested, testing is aimed at revealing the hidden persistent software errors. At this stage, a well-developed test plan should be available. It makes sense to reuse the existing test cases, rather than redesigning all new test cases, in retesting the program after it is corrected for any errors. Many modifications may occur during the maintenance phase where the software system is corrected, updated and fine-tuned.

The objective of regression testing is to ensure that no new errors have been introduced in the unmodified parts of the code due to the changes made [13]. Here, we would like to note that some existing papers in the literature also include testing the directly modified parts of the code as part of

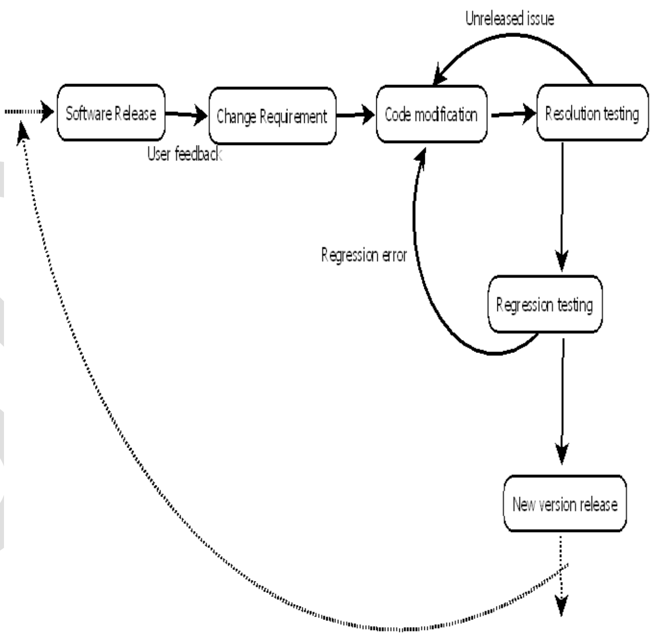

Fig. 1. Activities that take place during Software Maintenance and Regression Testing

regression testing. In our work, we consider testing the directly changed parts of the code as repeated execution of unit testing. Unit tests are re-executed to validate the modified parts of the code, while regression testing is carried out to revalidate the unchanged parts of the code that might have been affected by the code change. After testing is complete, a new version of the software is released, which then undergoes a similar maintenance cycle.

Regression testing is an expensive activity and is carried out after each modification to software [11, 12]. Regression Test Selection (RTS) is carried out to ensure that changes do not adversely affect unmodified portions of the software. It often accounts for almost half of the software maintenance costs [14]. To reduce regression testing costs, it is necessary to eliminate all those test cases that solely run the unaffected parts of the code, because they are unlikely to detect any bug. At the same time, it is also important to ensure that no test case that has the potential to detect a regression bug is overlooked. Accurate regression test selection is, therefore, considered to be an issue of considerable practical importance and has the potential to substantially reduce software maintenance costs [20]. Regression testing is carried out at various phases of software development life cycle such as, at unit, integration, system testing as well as during maintenance phase [8]. RTS techniques help to reduce the time and effort required to carry out regression testing.

RTS techniques based on analysis of both source code [1, 5, and 4] and model [7, 8, and 2] have been proposed in the literature for object-oriented software. Many RTS techniques first construct either the control flow [11, 4] or the dependency representation [5] of programs based on code analysis and then select test cases. These techniques compare the original and modified versions of the program model and select test cases that execute the affected model elements. In case of UML model-based RTS techniques, regression test cases are selected by comparing the original model with the model of the modified program [7, 8, and 1]. A problem with this approach is that models being abstraction after all, are often insensitive to minor code changes. In this context, we propose an RTS technique that considers control and data dependence information of object-oriented programs.

This paper is organized as follows: In Section 2, we discuss certain Basic concepts that provide the basic details needed to understand our approach. We explain our proposed approach in Section 3 and RTS in Section 4. We describe our empirical study in Section 5 and finally conclude the paper in Section 6.

II. BASIC CONCEPTS

In this section, we discuss certain basic concepts that underlie our approach to RTS for object oriented programs. We first present some definitions used in the context of regression test selection and then discuss a few models proposed for object oriented programs. Subsequently, we discuss some features of object oriented program that are relevant to regression test selection and also discuss a UML based RTS technique proposed by Naslavsky et al. [26] which we have used to compare our experimental results. For notational convenience, in the rest of the article we denote the original and the modified programs by P and P`, respectively. The initial test suite for P is denoted by T, and a test case in T is denoted by t.

A. *Concepts Related to Regression Test Selection*

In this section, we discuss a few important concepts and notations relevant to our work on regression test selection.

Rothermel and Harrold [21] have defined a set of metrics to evaluate the effectiveness of an RTS technique. However, their metrics were proposed in the context of procedural programs and do not consider specific characteristics of object oriented programs, such as the changed notion of correctness of an object oriented programs that also involves the notion of time.

In the context of object oriented programs, we argue that a more accurate metric of the efficacy of RTS is the number (or percentage) of test cases that are selected from those that failed when all the valid test cases in the initial test suite are run on the modified program. Thus, the percentage of failed test cases selected by an RTS technique can serve as a figure of merit.

1) Fault - Revealing Test Cases

Rothermel and Harrold [21] have defined a fault- revealing test case for a traditional program P as a test case $t \in T$ that can cause P to fail by producing in correct outputs for P. A test case $t \in T$ is said to be fault-revealing for programs P and P` if and only if it can cause P` to fail by producing an incorrect output or cause the output to be produced too late.

2) Modification - Revealing Test Cases

Rothermel and Harrold [21] have defined a modification-revealing test case as a test case $t \in T$ that produces different outputs for P and P`. A test case $t \in T$ is said to be modification-revealing for P and P` if and only if it produces different outputs when executed with P and P`, or if the outputs for P and P` are produced at different instants of time.

3) Relevant Regression Test Cases, Safety, and Precision

A test case $t \in T$ is relevant to a change if it executes those unmodified parts of P` which are affected due to data, control, or task execution dependencies. Therefore, all relevant test cases need to be executed during regression testing of P`.

B. *Program Models*

Graph models of programs have extensively been used in many applications, such as program slicing [22], reverse engineering [23], etc. Some of the popular procedural graph models reported in the literature include control flow graphs (CFG) [24], program dependence graphs (PDG) [25], and system dependence graphs (SDG) [12]. In the following, we briefly review an SDG graph model since it is related to our work.

System Dependence Graph (SDG) was first introduced by Horowitz et al. and was used to model procedural programs [12]. Later on, SDG was extended by Larsen and Harrold to model object-oriented programs [7].

An SDG is a directed, connected graph G = (V, E), consisting of a set V of vertices and a set E of edges. In the following, we describe the different types of edges and vertices in an SDG.

SDG Edges

- *Data dependence edge:* Data dependence edges are used to represent the data dependence relations.
- *Call edge:* A call edge is used to connect a call site to a method entry vertex.
- *Control dependence edge:* A control dependence edge is used to represent control dependence relations.
- *Summary edge:* A summary edge is used to represents the transitive dependence between actual-in and actual-out vertices**.**
- *Class member edge:* A class member edge is represent the membership relation between a class and its methods. A class entry vertex is connected to a method entry vertex by using a class member edge.
- *Parameter dependence edge:* Parameter dependence edges are represent passing values between actual and formal parameters in a method call.

SDG Vertices

- *Entry vertices:* In an SDG, classes and methods have entry vertices. A a method entry vertex represents an entry into a method and a class entry vertex represents an entry into a class.
- *Polymorphic choice vertex:* This is used to represent dynamic choice among the possible bindings in a polymorphic call.
- *Statement vertices:* Statements that are present in the methods are represented by statement vertices. There are two types of statement vertices: simple statement vertices and call vertices. Method call statements are represented by call vertices and all other statements such as assignments, conditionals loops and are represented by simple statement vertices.
- *Parameter vertices:* The parameter vertices are of four types. These include formal- in, formal-out, actual-in, and actual-out. The formal-in and formal-out vertices are created for each method entry vertex and actual-in and actual-out vertices are created for each call vertex and

A class is represented in an SDG by a Class Dependence Graph (ClDG) [5]. The root node of a ClDG is represented by a *class entry* vertex. Each method in a ClDG is represented by a procedure dependence graph [8]. Each method in a class has a *method entry* vertex.The *class entry* vertex is connected to the *method entry* vertex for each method in a class by *class member* edges.

In a ClDG, a method call is represented by a *call* vertex. For each method call vertex, the *actual-in* and *actual-out* vertices as well as *formal-in* and *formal-out* vertices are created for each called method. The *actual-in* parameter vertices are connected to the corresponding *formal-in* vertices in the

Example 1:

```
CE1: Class Calculator {
 S2: int a;
S3: int b;
E4: void set(int i,intj )
S5: a=i;
S6: b=j:
      }
E7: int add(){
S8: int result = a+b;
S9: return result;
     }
}
```

called method by *parameter-in* edges. The *formal-out* vertex of the called method is connected to the corresponding *actual-out* vertex at the calling method by a *parameter-out* edge. For a derived class, the representation of the base class method is reused for representing the inherited methods. Below program example shows a sample program and Figure 2 shows the SDG representation of this program.

C. *Effectiveness of a Regression Test Suite*

A regression test suite should include only that subset of original test suite that is likely to detect a regression error. To determine the effectiveness and quality of a regression test suite, Rothermel et al. have defined the concept of fault-revealing test cases for a program P [23].

D. *Program Slicing*

Program slice concept was first introduced by Weiser for debugging of programs [8]. A program slice consists of all those program statements that can affect the values computed at some point of interest called the slicing criterion [6, 12, and 7]. A forward program slice at a program point o with respect to a variable x contains all statements in the program, including conditionals that might be affected by any modifications to x at o [12]. A backward program slice at a program point o with respect to a variable x contains all statements in the program, including conditionals that might affect the value of x at o [12].

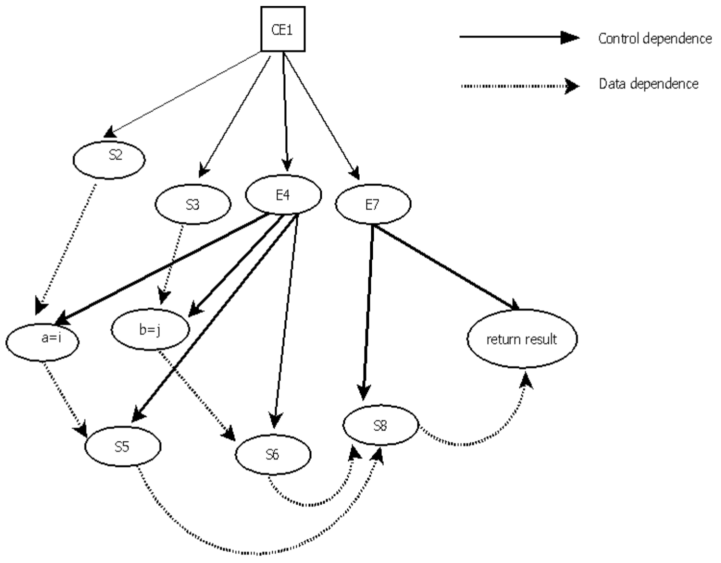

Fig. 2. SDG representation for the Program Example 1

E. *Naslavsky's UML-Based RTS Technique*

Naslavsky et al. [26] presented a model-based RTS technique that uses UML class and sequence diagrams for test selection. They transformed sequence diagrams of both the original and modified versions of a program into model-based control flow graphs. The traceability between test cases and the sequence diagrams is used to determine the elements of control flow graphs that are executed by each test case. Finally, the control flow graphs of both original and modified versions are analyzed and the test cases are selected using traceability information.

III. P-ReTEST: PROPOSED APPROACH

We have named our proposed approach for regression test case selection as P-ReTEST (Program Model Based Regression Test case Selector). Our technique selects regression test cases based on an analysis of control and data dependencies. In the following, we describe the important activities that are carried in P-ReTEST. As mentioned in Section I, the maintenance phase consists of multiple maintenance cycles, and in each maintenance cycle there can be many regression testing cycles. RTS is an important activity carried out in each regression testing cycle. The important steps of our approach P-ReTEST carried out in the first regression test selection cycle have been shown in Figure 3 using an activity diagram. As shown in Figure 3, the important activities in the first regression test selection cycle include constructing SDG model, collecting test coverage information and marking the test coverage information in SDG model are not repeated for subsequent regression test selection cycles in our approach. We now describe the different activities that are carried out during the first regression testing cycle.

The important steps in purposed approach is as follows as shown in figure 3

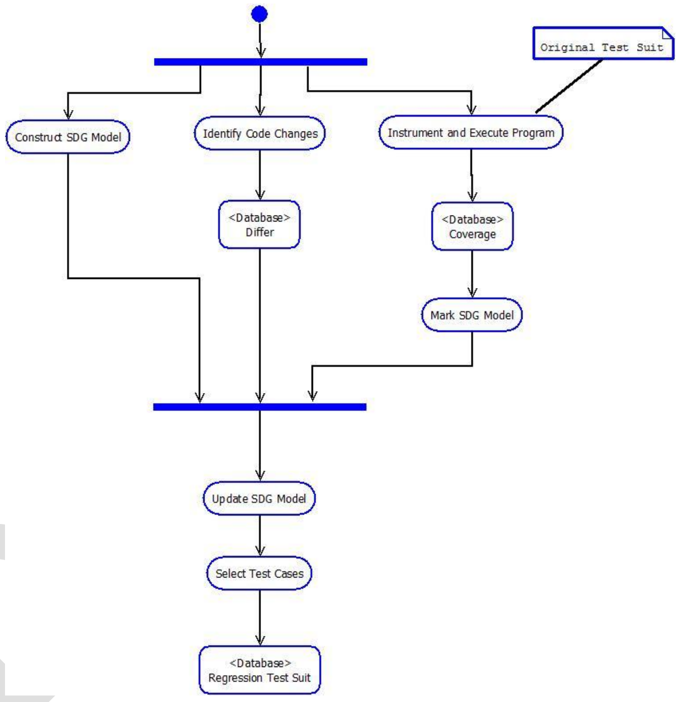

Fig. 3. Activity Diagram Representation

Step1: Construct SDG model: Very first, the SDG model for the original program P will construct using a technique specified by Larsen and Harrold [20].
Step2: Identify changes: The changes between P and the modified program P' will be identified through analysis. These identified statement-level changes will be kept in a file named as Differ. Each entry in Differ file contains the changed statement in P', the line number in P or P', the name of the method and the class to which the changed statement belongs. This is shown by the data store Differ in Figure 3.
Step3: Instrument and execute the program: In this step, original program P will be instrumented by inserting print statements and instrumentation will be done at basic block level. The print statements will insert to collect test coverage information. The instrumented code will be executed with the original test suite T to generate information, which statements are executed for each test case. The test coverage information generated in this step denoted by Coverage file in figure 3 and is saved for later processing.
Step 4: Mark the SDG model: The test coverage information will be marked on SDG model.
Step5: Update the SDG model: The model constructed for original program P will update during each regression testing cycle to make it correspond to the modified program P' using information stored in file Diff.
Step6: Select test cases: In this step, regression test cases will select based on analysis of SDG model.

A. *Types of Program Changes*

An arbitrary change to a program could be any one of the following three types: (1) addition of a statement, (2) deletion of a statement, or (3) modification of a statement. A change to a program P could be confined to a single line or could span multiple lines. A change to P might require addition and deletion of some nodes and edges of the corresponding SDG model. Any arbitrary modification could be considered to be composed of a deletion operation followed by an addition operation. Therefore, in our work, we assume that addition and deletion are the only two basic change operations. In the following, we identify the changes to the SDG model required to reflect the changes caused due to the two basic program change operations.

A single statement-level change could affect the dependency relations among various elements of a program in subtle ways. In the following, we elaborate how the control flow and dependency relations are affected due to the two basic types of code changes: addition and deletion.

-Addition of Statements: Adding new statements to P requires creating new nodes and edges in the SDG model M. The additional edges created could be of types control flow, control or data dependence, parameter-in, etc. It may also be required to delete certain existing control flow and dependency edges during edge creation.

-Deletion of Statements. Deletion of one or more statements could affect the dependencies existing among certain other statements, for example, if a statement that defines a variable is deleted, it could lead to a wrong evaluation of a predicate which uses that variable. Therefore, before actual deletion of statements, it is important to identify and mark all those program elements as affected which are data dependent on the deleted statement before actual deletion. Before a statement (i.e., one or more nodes) is deleted, first the other nodes in M that are data or control dependent on the deleted node(s) are identified and are marked as affected. Then, the node(s) in M corresponding to the deleted statement are deleted. The different edges which are incident on or emanate from the node(s) corresponding to the deleted statement are also deleted. In addition, new data- and control-dependency edges can get created on account of the modified dependency relationships.

B. *Regression Test Selection*

The set of selected regression test cases (*TREG*) can be expressed as:

$$T_{REG} = T_{DEP}$$

Where, T_{DEP} denotes the test cases selected through control and data dependence analysis and dependencies due to object-relations.

C. *Determination of T_{DEP}*

Regression test cases, T_{DEP}, are determined based on an analysis of the constructed SDG model. To select T_{DEP}, we first compute the forward slice on updated marked SDG model. Our test case selection algorithm is based on graph reachability algorithm proposed by Horwitz[19], where each marked model element that are tagged during *Update SDG model* step, is taken as the selection criterion.

Our Proposed Algorithm 1 selects test cases from SDG model. Algorithm takes updated SDG model denoted by *M* and the set of tagged nodes denoted by *Tagged* obtained during *update SDG model* step as input, and produces the selected set of regression test cases as the output, T_{DEP}. Algorithm computes the set of all affected nodes denoted by *Affectednodes* on basis of data and control dependencies or dependencies arising due to object-relations such as inheritance, the steps are given in lines 2 to 5 in Algorithm. After all the affected nodes in SDG have been identified through forward slicing, the test cases that execute these affected nodes are selected for regression testing. This is done by traversing the SDG model and visiting each node in *Affectednodes* to determine the test cases that execute these affected nodes.

V. EXPERIMENTAL STUDIES

We have named our prototype tool as P-ReTEST (Program Model Based Regression TEST case selector).We have implemented a tool based on our proposed approach for RTS.

A. *P-ReTEST*

A Prototype Implementation of RTS P-ReTEST has been developed using the programming language Java on a Microsoft Windows 7 environment. The code size of P-ReTEST is approximately 12 KLOC, excluding the external packages that are used in implementation of RTS technique. The user interface of P-ReTEST is developed using Java Swing. In the following, we describe the various open source software packages used to implement RTS.

Algorithm 1: Pseudocode to select Regression Test Cases

```
Input:    M, Tagged
1.  SDGSELECT(M, Tagged, $T_{DEP}$)
2.  For each node n in Tagged do
3.  Find the node that are data and control dependent
4.  Affectednode = NULL
5.  Affectednode = Affected node U{all nodes that are
    data and control dependent }
6.  end
```

```
7.  if Affected node ≠ ϕ then
8.  for each node n € Affectednode do
9.  Add all test cases that execute n to $T_{DEP}$
10. End
```

Output: T_{DEP}

–

B. *Open source software packages used*

We have developed the tool P-ReTEST using the following open source software packages: Eclipse [3], Cygwin [1] and Graphviz [4]. We have used eclipse as an IDE and CCygwin is used to provide Linux Environment on window OS to run Linux command using a Java Program.. To graphically visualize the SDG model constructed by P-ReTEST, we have used Graphviz.

C. *Experiments*

In this section, we discuss the specific experimentation carried out by us using P-ReTEST to measure the effectiveness of our approach. We have used the following programs namely, Climate Controller, Vending Machine, Automated Teller Machine, and Power Window Controller in our experimentation. The size of the considered programs range from 400 to 900 LOC as given in Table 1. Each of the considered programs had on an average of 25 test cases. For each program, we created several modified versions. We have considered the different types of modifications that are made in each version of a program from Ren et al. [18]. We tested each modified version of a program by running the original test cases on each modified version of a program to note the number of test cases failed after modification. Then, each time the test cases were selected using P-ReTEST and also from Naslavsky's UML based analysis. We repeated the experiment for each modified version of each considered program in order to remove any bias introduced in the results due to a specific type of change. To measure the effectiveness of our RTS technique, we have calculated the average percentage of fault- revealing test cases selected by P-ReTEST and by Naslavsky's UML model analysis.

D. *An Evaluation of the Effectiveness of P-ReTEST*

The aim of our experimental studies using P-ReTEST was to evaluate the performance and effectiveness of our RTS approach. An intuitive and appealing metric for evaluating the effectiveness of an RTS technique is the size of the selected regression test suite. Obviously, it is desirable to have this number as small as possible. However, for effective RTS, it is more important for a technique not to miss out selecting any fault-revealing test cases, and at the same time, to minimize instances of false positives. Therefore, we have defined a new metric called fault-revealing effectiveness. In the following, we briefly describe these two metrics with which we evaluated the effectiveness of P-ReTEST.

Percentage of Test-Cases Selected for RTS (ϒ) - This measure indicates the size of the regression test suite as a percentage of the initial test suite.

Fault-Revealing Effectiveness (Ω) - The fault-revealing effectiveness metric can be defined as the percentage of test cases selected by an RTS technique from the set of test cases that fail when the valid test cases in the initial test suite are run. That is, the fault-revealing effectiveness of the test suite selected by a safe RTS technique is equal to 100%, that is, it is equal to that of the initial test suite.

Table I Summary of Regression Test Selection Results

Program	**Number of LOC**	**Number of test cases**	**Percentage of test cases selected by - P-ReTEST**	**Percentage of test cases selected by Naslavsky's Approach**	**Percentage Increase**
Climate Controller	510	32	45	28	53.66
Vending Machine	451	21	46	34	34.11
Automated Teller Machine	603	22	58	42	32.77
Power Window Controller	742	26	68	47	33.53

E. *Result*

In this section, we describe the results obtained from experimental studies carried out by us to determine the effectiveness of our RTS technique.

Table I and Table II summarize our experimental results. Table I summarizes the percentage of test cases selected by our approach and Naslavsky's approach. In Table I, the example programs used in our experimental studies is given in column 1 and column 2 shows the lines of code (LOC) for each of our example programs. In column 3, we list the total number of test cases in the initial test suite and the percentage of test cases selected while executing the entire test suite on the modified programs by P-ReTEST and by Naslavsky's approach is reported in column 4 and column 5 respectively. The percentage increase in the regression test suite size is given in column 6. P-ReTEST on an average selects 38.21 % more than the only Naslavsky's approach. This increase may be due to the fact that, our approach selects test cases based on code analysis.

Table II summarizes the average percentage of fault-revealing test cases selected by both approaches. In Table II, the test cases failed is given in column 2. The average percentage of fault-revealing tests selected by P-ReTEST and Naslavsky's approach is given in columns 3 and 4 respectively. The results show that P-ReTEST selects all the fault-revealing test cases and the percentage of fault-revealing test cases selected by P-ReTEST is on an average of 27.89 % higher than a Naslavsky's UML -based analysis.

Table II Summary of Quality Results

Program Name	**Percentage of test cases failed**	**Percentage of fault-revealing tests selected by P-ReTEST**	**Percentage of fault-revealing tests selected from Naslavsky's UML -based analysis**
Climate Controller	29	100	75
Vending Machine	20	100	74
Automated Teller Machine	21	100	78
Power Window Controller	19	100	72

F. *Analysis*

The results of Table I have been presented in the form of a bar graph in Figure 4. In the figure 4, the y-axis shows the percentage of selected test cases while the labels on the x-axis represent the different programs. It can be observed from Table I and Figure 4 that P-ReTEST selected around 45% to 68% of test cases for regression testing of the modified programs. Considering the results for all the programs, the number of test cases selected by P-ReTEST was on average 37.34% greater than Naslavsky's approach [26]. This increase can be explained by the fact that, in addition to control dependence, our approach also selects test cases based on system dependencies that are ignored by Naslavsky's approach.

The results of Table II have been presented as a bar graph in Figure 5. In the figure, the y-axis shows the percentage of failed test cases selected while the labels on the x-axis represent the different programs. The results show that P-ReTEST is able to select all the fault-revealing test cases present in T. In other words, the regression test suite selected by P-ReTEST has the same fault-revealing effectiveness Ω as the initial test suite. The fault-revealing effectiveness of Naslavsky's approach is lower by 26.36% on average compared to ReTEST.

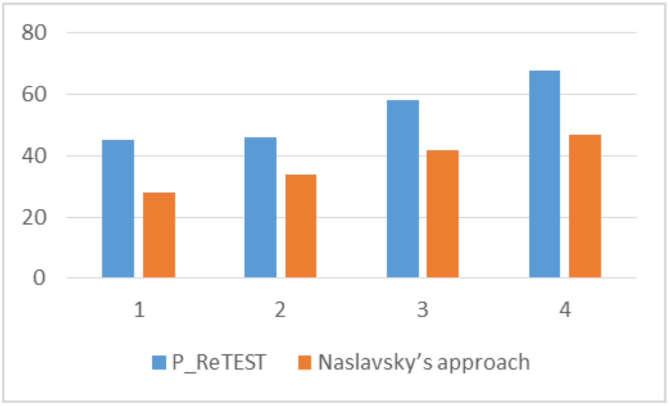

Fig. 4. Percentage of regression test cases selected (ϒ)

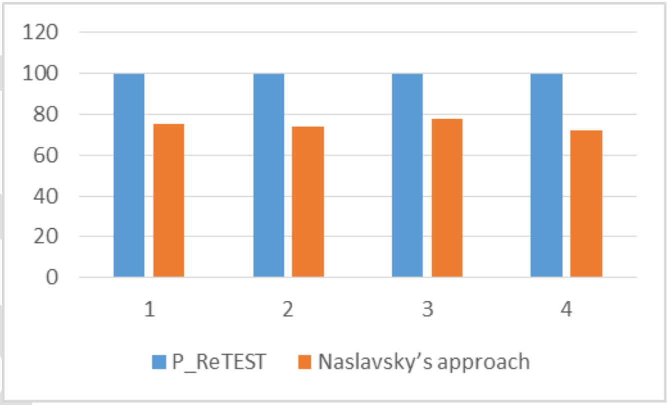

Fig. 5. A comparison of the fault-revealing effectiveness (Ω) of P_ReTEST and Naslavsky's approach.

CONCLUSION

We have presented an approach for regression test selection of object-oriented programs that selects test cases by analyzing source code. We have applied the proposed RTS technique to small example programs to prove the applicability of our approach. The results of our study show the effectiveness in selecting more fault-revealing test cases from the original test suite. In our empirical studies, we observe an average increase of 26.36% selection of fault-revealing test cases in P-ReTEST as compared to Naslavsky's UML model based analysis.

REFERENCES

[1] http://www.cygwin.org/.
[2] http://www.bugzilla.org/.
[3] http://www.eclipse.org/.
[4] http://www.graphviz.org/.
[5] R. V. Binder, "Testing Object-Oriented Systems: Models", Patterns, and Tools, Addison-Wesley, (2003).

[6] L. Briand, Y. Labiche and S. He, “Automating Regression Test Selection Based on UML Designs”, Journal of Information and Software Technology, (2009), pp. 16-30.

[7] H. Do, S. Mirarab, L. Tahvildari and G. Rothermel, “The Effects of Time Constraints on Test Case Prioritization:A Series of Controlled Experiments”, IEEETransactions on Software Engineering, vol. 36, no. 5, (2010), pp. 593-617.

[8] M. Harrold, J. Jones, T. Li, D. Liang and A. Orso, “Regression Test Selection for Java Software”, In Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications, (2001), pp. 312-326.

[9] S. Horwitz, T. Reps and D. Binkley, “Interprocedural Slicing Using Dependence Graphs”, Transactions on Programming Languages and Systems, vol. 12, no. 1, (1990), pp. 26-60.

[10] Y. K. Jang, M. Munro and Y. R. Kwon, “An Improved Method of Selecting Regression Tests for C++ Programs”, Journal of Software Maintenance: Research and Practice, vol. 13, (2001), pp. 331-350.

[11] G. Kapfhammer, “The Computer Science Handbook”, Chapter Software Testing, (2004), CRC Press, Boca Raton, FL.

[12] D. Kung, J. Gao, P. Hsia, Y. Toyoshima and C. Chen “Firewall Regression Testing and Software Maintenance of Object - Oriented Systems”, Journal of Object-Oriented Programming, (1997).

[13] H. Leung and L. White. Insights into regression testing. In Proceedings of the Conference on Software Maintenance, pages 6069, 1989.

[14] N. Wilde and R. Huitt Maintenance support for object-oriented programs, IEEE Transactions on Software Engineering,December 1992.

[15] Mr. Rohit N. Devikar, Prof. Manjushree. D. Laddha, “Automation of Model-based Regression Testing”, International Journal of Scientific and Research Publications, Volume 2, Issue 12, December 2012.

[16] G. Kapfhammer. The Computer Science Handbook, chapter on Software testing. CRC Press, Boca Raton, FL, 2nd edition, 2004.

[17] S. Yoo, M. Harman, “Regression Testing Minimization, Selection and Pri- oritization: A Survey” Softw. Test. Verif. Reliab. 2007,Wiley InterScience.

[18] X. Ren, O. C. Chesley and B. G. Ryder, “Identifying Failure Causes in Java Programs: An Application of Change Impact Analysis”, IEEE Transactions on Software Engineering, vol. 32, no. 9, (2006), pp. 718 – 732.

[19] A. Orso, N. Shi, and M. Harrold. Scaling regression testing to large software systems. In Proceed- ings of the 12th ACM SIGSOFT Twelfth Interna- tional Symposium on Foundations of Software Engineering, pages 241251, November 2004.

[20] GUAN, J., OFFUTT, J.,AND AMMANN, P. 2006. An industrial case study of structural testing applied to safety-critical embedded software. In Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering. ACM, New York, NY, 272–277.

[21] ROTHERMEL, G.AND HARROLD, M. 1996. Analyzing regression test selection techniques. IEEE Trans. Softw. Eng. 22, 8, 529–551.

[22] LIANG, D.AND HARROLD, M. 1998. Slicing objects using system dependence graphs. In Proceedings of the International Conference on Software Maintenance. IEEE Computer Society, Los Alamitos, CA,358– 367.

[23] CLEVE, A., HENRARD, J., AND HAINAUT, J. 2006. Data reverse engineering using system dependency graphs. In Proceedings of the 13th Working Conference on Reverse Engineering. IEEE Computer Society, Los Alamitos, CA, 157–166.

[24] AHO, A., SETHI, R.,AND ULLMAN, J. 2008. Compilers: Principles,Techniques and Tools 2nd Ed. Dorling Kinder- sley (India) Pvt Ltd.

[25] FERRANTE,J.,OTTENSTEIN,K.,ANDWARREN,J.1987.The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. 9, 3, 319–349.

[26] L. Naslavsky, H. Ziv and D. J. Richardson, “A Model-Based Regression Test Selection Technique”, In 25th IEEE International Conference onSoftware Maintenance, (2009). Edmonton, Alberta, Canada.