International Journal of Hybrid Information Technology
Vol.8, No.6 (2015), pp.341-356
http://dx.doi.org/10.14257/ijhit.2015.8.6.33

A Program Model Based Regression Test Selection Technique For D Programming Language

Nitesh Chouhan[1], Dr. Maitreyee Dutta[2] and Dr. Mayank Singh[3]

[1]Assistant Professor, [2]Associate Professor, [3]Associate Professor,
[1]Department of IT, MLVTEC Bhilwara, India
[2]Department of CSE, NITTTR, Chandigarh, India
[3]Department of CSE, KEC Ghaziabad, UP, India
[1]niteshchouhan_9@yahoo.com, [2]dr_maitreyee@gmail.com,
[3]mayanksingh2005@gmail.com

Abstract

Software testing can be stated as the process of validating and verifying that a computer program, application and product [1]. Software testing can also provide an objective, independent view of the software to allow the business to appreciate and understand the risks of software implementation. Software testing, depending on the testing method employed, can be implemented at any time in the development process. Traditionally most of the test effort occurs after the requirements have been defined and the coding process has been completed. Testing can never completely identify all the defects within software. A primary purpose of testing is to detect software failures so that defects may be discovered and corrected. Testing cannot establish that a product functions properly under all conditions but can only establish that it does not function properly under specific conditions. There are many approaches to software testing. Reviews, walkthroughs, or inspections are referred to as static testing, whereas actually executing programmed code with a given set of test cases is referred to as dynamic testing.

Regression testing is an important but expensive software maintenance activity performed with the aim of providing confidence in modified software. Regression test selection techniques reduce the cost of regression testing by selecting test cases for a modified program from a previously existing test suite. Regression testing is done every time when a program is modified to ensure that the modifications do not introduce new bugs into previously validated code. Regressions Testing can be done by collectively perform Regression Test Selection, Test Minimization and Test Case Priotrization Technique.

An important research problem, in this context, is the selection of a relevant subset of test cases from the initial test suite. Regression test selection (RTS) techniques minimize both the regression testing time and effort. Regression test selection (RTS) techniques select a subset of valid test cases from an initial test suite (T) to test that the affected but unmodified parts of a program continue to work correctly. Use of an effective regression test selection technique can help to reduce the testing costs in environments in which a program undergoes frequent modifications.

D is a new programming language. This is an object-oriented, imperative, multi-paradigm system programming language. Regression testing on D programming language still untouched by researchers. Our research attempts to bridge this gap by introducing a techniques to revalidate D programs. A framework is proposed which automates both the regression test selection and regression testing processes for D programming language. As part of this approach, special consideration is given to the analysis of the source code of D language. In our approach system dependence graph representation will be used for regression test selection for analyzing and comparing the *code changes of original and modified program. First we construct a system dependence graph of the original program from the source code. When some modification is executed in a program, the constructed graph 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. 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 UML based technique while incurring about 37.34% increase in regression test suite size.*

Key Words: *System Dependence Graph, Regression testing, Regression Test Selection (RTS) and Control Flow Graph*

1. Introduction

Programming languages are used for controlling the behavior of a machine (often a computer). There are thousands of programming languages and new ones are created every year. Few languages ever become sufficiently popular that they are used by more than a few people, but professional programmers may use dozens of languages in a career. D appeals to programmers who are interested in writing high performance code, want a C++ style language, but need a language that is much easier to master with support for modern techniques like automatic memory management, modules etc.

Regression testing is an expensive activity and is carried out after each modification to software [11, 12]. 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]. 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. 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. Regression testing on object oriented programming language still not touched by researchers efficiently. Our research attempts to bridge this gap by introducing a techniques to revalidate object oriented programs of Java.

RTS techniques based on analysis of both source code [1, 5, 4] and model [7, 8, 2] have been proposed in the literature for object-oriented program. 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, 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 D 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. We describe our empirical study in Section 4 and finally conclude the paper in Section 5.

2. Basic Concepts

In this section, we discuss certain basic concepts that underlie our approach to RTS for object oriented programs. We first present introduction about Software Testing, Regression Testing, D programming language and 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 paper 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.

2.1 D Language

During the past few years, programming languages have come a long way. In comparison to the dawn of UNIX and C, when compiled languages were just getting their start, today's world is full of languages with all sorts of goals and features. In this paper, we focus on one such language, D from Digital Mars. D is a general purpose systems and applications programming language. It is a higher level language than C++, but retains the ability to write high performance code and interface directly with the operating system API's and with hardware. D's features include the lack of a preprocessor, a garbage collector, flexible first-class arrays, contracts, inline assembler and more. It isn't unusual for a D program to have 30% less source code than the equivalent C++, yet run at the same speed or faster [27]. It's simply faster to develop code in D, and faster to get it debugged. D is well suited to writing medium to large scale million line programs with teams of developers. It is easy to learn, provides many capabilities to aid the programmer, and is well suited to aggressive compiler optimization technology. It is a promising language that is able to supply many different needs in the programming community. Features such as arrays, SH syntax and type inference make D comparable to languages, such as Ruby and Python, in those regards, while still it is open for low-level system programmers with the inline assembler and other features. It brings in features of imperative languages, such as Lisp, with the lazy storage class, which drastically speeds up efficiency. The language is relatively stable, with the occasional new features or changes added in. There are two major versions of the language - D1 and D2. D1 is stable (will undergo no other changes), and D2 is a major revision of the language that sacrificed some backwards compatibility, and for adding a few crucial features related to generic programming. There are also two essential D libraries, the official -Phobos, and a community-driven library called Tango. Tango, designed for D1, is being ported to D2, and Phobos is undergoing major changes and additions to take full advantage of D2's capabilities. Last but definitely not least, two windowing libraries complete the language's offering quite spectacularly. The mature library DWT is a direct port of Java's SWT. A newer development is that the immensely popular Qt Software windowing library has recently released a D binding. D supports these programming paradigms of C, C++,JAVA- imperative, object-oriented, meta programming, functional and concurrent. 1000 random number has been generated using same program logic explain in algorithm 1 in different language i.e C, java, C# and D language[27]. And calculate the how much time is required to run the program in different language which is shown in table 1.

Algorithm#1

```
IM 139968
IA 3877
IC 29573
```

```
last 42
gen_random(double max) {
last = (last * IA + IC) % IM;
return( max * last / IM );}
 int N =1000;
while (N--) {
result = gen_random(100.0);}
printf( result);
```

Table 1. Execution Time for Random Number Genrator

Language	Execution time in second
C	2.0879
Java	0.141
C Sharp	0.14
D	0.119

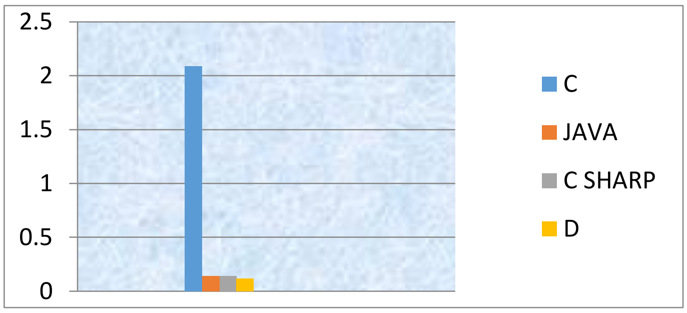

Figure 1. Performance Chart

2.2. Software Testing

We are all human beings and therefore it is natural for us to make mistakes. It is generally accepted, and also noted that for programmers it is natural to introduce bugs into software during the development process. Beizer also high- lights the statistic that says for every 100 statements written by a good programmer there are still 1-3 bugs. In our modern life software is used in almost everything that surrounds us. Take, for example, home appliances like microwave ovens or washing machines, cell phones; take a car that alone might have up to several dozen computers installed in it, and so on. Bugs revealed in some of those would make one feel upset, but in others it might cost human lives. And that is why software testing is an important stage of the development process that cannot be omitted or ignored. There are many published definition of software testing, however all definitions boil down to essentially the same thing: software testing is the process of executing software in controlled manner in order to answer the question "Does the software behave as specification".

Software testing often used in associated with the term verification and validation. Verification is checking and testing of the item for conformance and consistency. Software testing is the one kind of verification which also uses technique reviews, analysis audit, inspections and walkthrough. Validation is the process of checking that what has been specified is what the user actually wanted.

2.3 Regression Testing

Regression testing (also referred to as program revalidation) is carried out to ensure that no new errors (called regression errors) have been introduced into previously validated code (i.e., the unmodified parts of the program) [2]. Let P be an application program and P′ be a modified version of P. Regression testing is done after certain changes have been introduced to a piece of software or program P. It is performed during the maintenance phase of the evolution of the program. Regression testing's purpose is to verify that modified software preserves the expected behavior and does not introduce errors. Even if the change is very small, the impact caused by that change can be very tangible and would affect the behavior of the software, perhaps by introducing new unexpected faults. Although regression testing is usually associated with system testing after a code change, regression testing can be carried out at unit, integration or system testing levels.

The sequence of activities that take place during the maintenance phase after the release of software is shown in Figure 2. The Figure 2 shows that after software is released, the failure reports and the change requests for the software are compiled, and the software is modified to make necessary changes. Resolution tests are carried out to verify the directly modified parts of the code, while regression test cases are 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 [2].

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.

Software maintenance is defined as the performance of those activities required to keep a software system operational and responsive after it is accepted and placed into production.

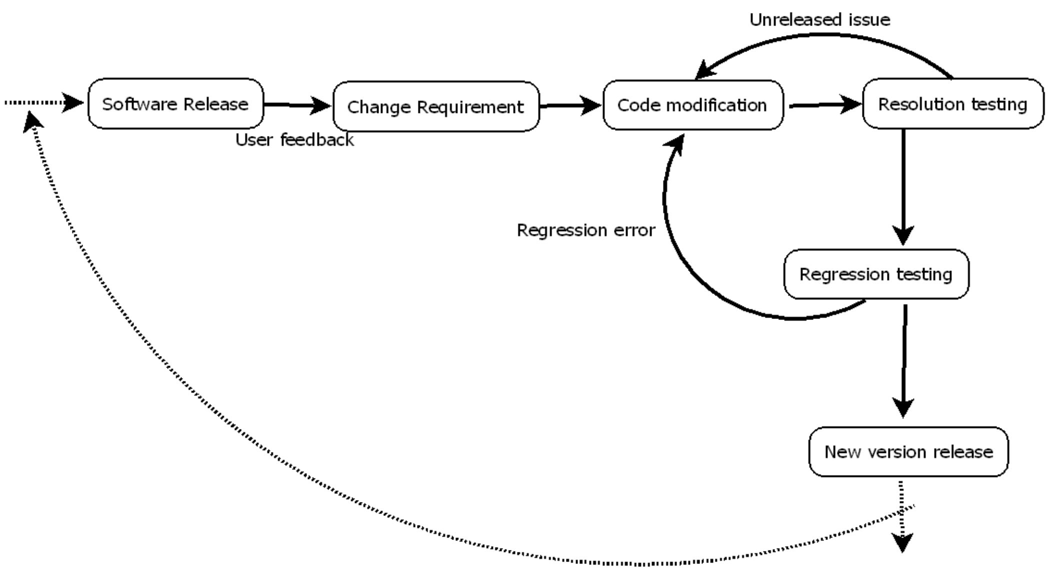

Figure 2. Activities that Take Place during Software Maintenance and Regression Testing

2.3.1. Types of Regression Testing

Two types of regression testing can be identified based on the possible modification of the specification [2]: Progressive and Corrective Regression testing. Progressive regression testing involves a modified specification. Whenever new enhancements or new data requirements are incorporated in a system, the specification will be modified to reflect these additions. In most cases, new modules will be added to the software system with the consequence that the regression testing process involves testing a modified program against a modified specification.

In corrective regression testing, the specification does not change. Only some instructions of the program and possibly some design decisions are modified. This has important implications because most test cases in the previous test plan are likely to be valid in the sense that they correctly specify the input-output relation.

2.3.2 Regression Test Cases

Leung and White categorize test cases into five classes [2]. The first three classes consist of test cases that already exist in test suit T of original program P.

1. Reusable: Reusable test cases only execute the parts of the program that remain unchanged between two versions, i.e. the parts of the program that are common to P and P`. It is unnecessary to execute these test cases in order to test P`; however, they are called reusable because they may still be retained and reused for the regression testing of the future versions of P.

2. Resettable: Resettable test cases execute the parts of P that have been changed in P`. Thus resettable test cases should be re-executed in order to test P`.

3. Obsolete/Redundant: Test cases can be rendered obsolete because:

- ➢ Their input/output relation is no longer correct due to changes in specifications,
- ➢ They no longer test what they were designed to test due to modifications to the program,
- ➢ They are 'structural' test cases that no longer contribute to structural coverage of the program.

The remaining two classes consist of test cases that have yet to be generated for the regression testing of P`.

4. New-Structural: New-structural test cases test the modified program constructs, providing structural coverage of the modified parts in P`.

5. New-Specifications: New-specifications test cases test the modified program specifications, testing the new code generated from the modified parts of the specifications of P`.

2.3.3 Regression Testing Technique

The three major branches include test suite minimization, test case selection and test case prioritization [11].

Test Suite Minimization is a process that seeks to identify and then eliminate the obsolete or redundant test cases from the test suite.

Test Case Selection deals with the problem of selecting a subset of test cases that will be used to test the changed parts of the software.

Test Case Prioritization concerns the identification of the 'ideal' ordering of test cases that maximize desirable properties, such as early fault detection.

2.4 Program Models

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.

Let V be the set of all node types of an SDG. Then, V can be expressed as follows.

V={Ve, Vs, Vp }, where each member of these represents a particular node type. In the following, we explain the different types of nodes in an SDG.

• Entry vertices (Ve): In an SDG, classes and methods have entry vertices. A method entry vertex represents an entry into a method and a class entry vertex represents an entry into a class.

• Statement vertices (Vs): 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 (Vp): 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.

Let E denote the different types of edges of an SDG. It can be expressed as E ={Ecd, Edd, Ece, EPin, EPout, ESum}, where each member of these E represents a particular edge type. In the following, we explain the different types of edges of an SDG.

• Control (Ecd)- and data (Edd)-dependence edges represent the control and data dependence relationships among the nodes of an SDG, respectively.

• Call edges (Ece) link the call-site nodes with the corresponding procedure entry nodes.

• Parameter-in edges (EPin) connect the actual-in nodes with the respective formal-in nodes.

• Parameter-in edges (EPin) connect the actual-in nodes with the respective formal-in nodes.

• Parameter-out edges (EPout) connect the formal-out nodes with the respective actual-out nodes.

• Summary edges (ESum) are used to represent the transitive dependencies that arise due to function calls. A summary edge from an actual-in node a to an actual-out node b is constructed if the value associated with b can get affected by the value associated with the node a due to control or data dependence, that is, a summary edge from a to b is constructed if there exists either a control or data-dependence edge from the corresponding formal-in node a` to the formal-out node b`. Below program example shows a sample program and Figure 3 shows the SDG representation of this program.

Example 1:

```
CE1: Class Cal {
 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;
     }
 }
```

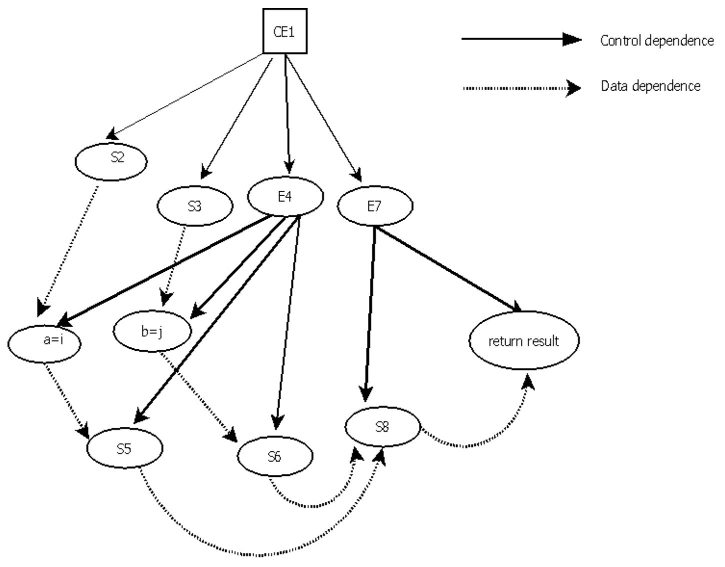

Figure 3. SDG representation for the Program Example 1

2.5. 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].

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∈ T that can cause P to fail by producing in correct outputs for P. A test case t ∈ 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.

2.6 Program Slicing

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, 7].

2.7 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 control flow graphs of both original and modified versions are analyzed and the test cases are selected using analysis.

2.8 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 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.

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, 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.

3. 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 shown in Figure 4, 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 4

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.

Step3: Instrument and execute the program: In this step, original program P will be instrumented by inserting print statements. The print statements will insert to collect test coverage information. The original source code P will be executed with the original test suite T to generate information, which statements are executed for each test case.

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.

Step6: Select test cases: In this step, regression test cases will select based on analysis of SDG model.

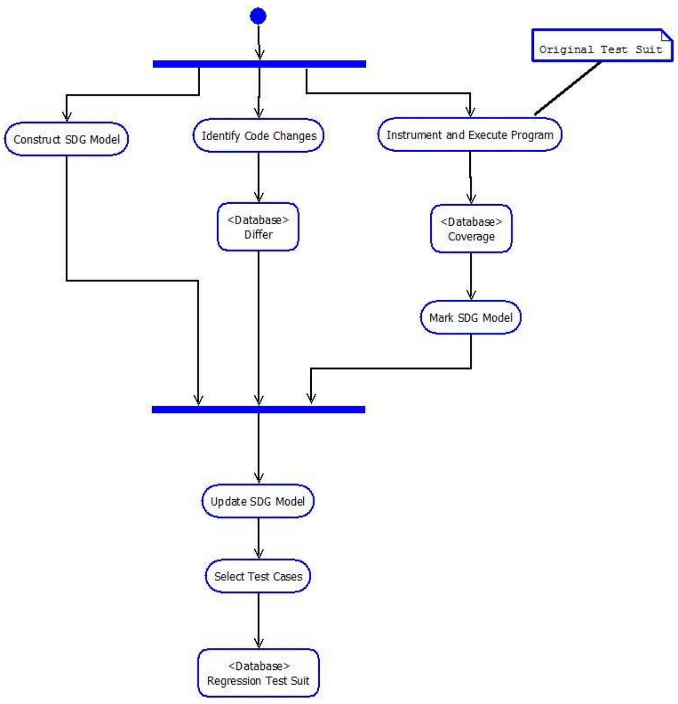

Figure 4. Activity Diagram Representation

3.1. Determination of Regression Test Cases

Regression test cases, TReg, are determined based on an analysis of the constructed SDG model.Our Proposed Algorithm 2 selects test cases from SDG model. Algorithm takes updated SDG model denoted by M and the set of tagged nodes denoted by Tag obtained during update SDG model step as input, and produces the selected set of regression test cases as the output, TReg. Algorithm computes the set of all affected nodes denoted by Affected nodes on basis of data and control dependencies, the steps are given in lines 2 to 5 in Algorithm1. After all the affected nodes in SDG have been identified, 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 Affected nodes to determine the test cases that execute these affected nodes.

Algorithm 2 to select Regression Test Cases

```
Input:   M, Tag
1.       SDGSELECT(M, Tagged, $T_{Reg}$)
2.       For each node n in Tagg do
3.       Find the node that are data and control dependent
4.       Affected = NULL
5.       Affected = Affected  U{all nodes that are data and control dependent }
6.       end
7.       if  Affected $\neq \phi$ then
8.       for each node n € Affected do
9.       Add all test cases that execute n to $T_{Reg}$
10.      End
Output: $T_{Reg}$
```

Where, TReg denotes the test cases selected through control and data dependence analysis and dependencies due to object-relations. It is also called regression test cases.

4. 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.

4.1 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 10 KLOC. 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.

4.2 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 Cygwin is used to provide Linux Environment on window OS to run Linux command to find out difference using a Java Program. To graphically visualize the SDG model constructed by P-ReTEST, we have used Graphviz.

4.3 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, Addition, Deletion, Looping, and Demo in our experimentation. Where, changes in Addition program is done by adding a statement, changes in Deletion program is done by deleting a statement, changes in Looping program is done by adding one more for loop, changes in Demo program is done by modification in a if else statement. Each of the considered programs had on an average of 20 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.

(a) Certified

```
public class ForLoop
{
 public main()
 {
 int c;
 for (c = 1; c <= 10; c++) {
 println(c);
    }
  }
}
```

(b) Modified

```
public class ForLoopModified
{
public main()
{
 int c,d;
for (c = 1; c <= 10; c++)
{
for(d=0; d<5; d++)
{
println(c);
             }
          }
       }
}
```

Figure 5. Source Code of Looping

Then, each time the test cases were selected using P-ReTEST and also from Naslavsky's UML based analysis. 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.

Source code of a Looping program and after modification in a statement in this program is shown in Figure 5. A snapshot of the SDG model using Graphviz for the both program is shown in Figure 6.

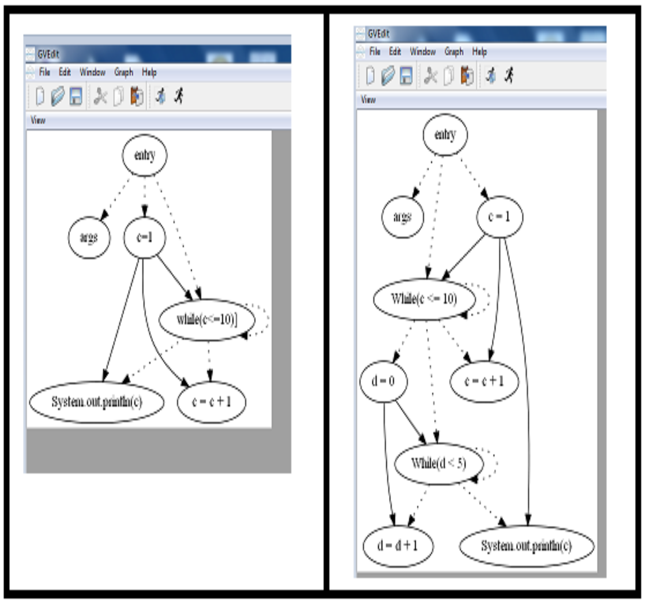

Figure 6. Graphviz GENERATEd System Dependence Graph

4.4. 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. Obviously, it is desirable to have regression test cases 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. 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 75%, that is, it is equal to that of the initial test suite.

4.5 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 2. Summary of Regression Test Selection Results

Program	Number of test cases	Percentage of test cases selected by - P-ReTEST	Percentage of test cases selected by Naslavsky's Approach	Percentage Increase
Addition	31	45	28	53.66
Deletion	21	46	34	34.11
Looping	20	48	32	32.77
Demo	26	68	47	33.53

Table 2 and Table 3 summarize our experimental results. Table 2 summarizes the percentage of test cases selected by our approach and Naslavsky's approach. In column 2, 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 3 summarizes the average percentage of fault-revealing test cases selected by both approaches. In Table 3, 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.

4.5 Analysis

The results of Table 2 have been presented in the form of a bar graph in Figure 7. In the Figure 7, 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 2 and Figure 7 that P-ReTEST selected around 45% to 68% of test cases for regression testing of the modified programs.

Table 3. 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
Addition	29	75	54
Deletion	20	74	52
Looping	21	78	56
Demo Controller	19	72	50

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.

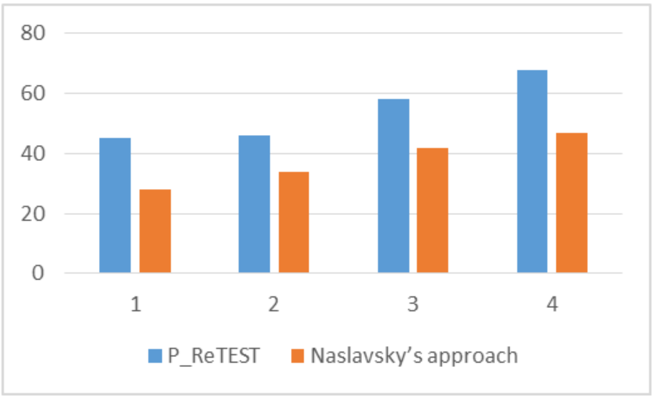

Figure 7. Percentage of regression test cases selected (ϒ)

The results of Table 3 have been presented as a bar graph in Figure 8. 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.

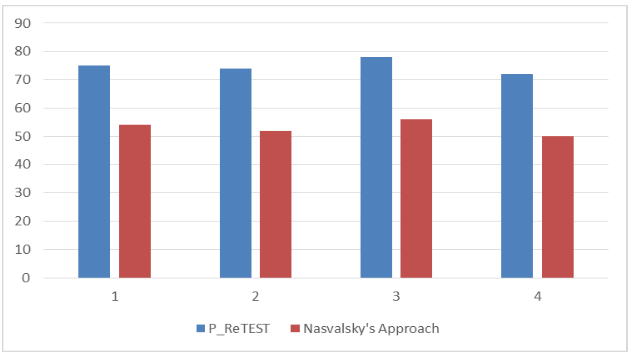

Figure 8. A comparison of the fault-revealing effectiveness (Ω) of P_ReTEST and Naslavsky's approach

5. 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", IEEE Transactions 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 Proceedings of the 12th ACM SIGSOFT Twelfth International 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.19 87. 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 on Software Maintenance, (2009). Edmonton, Alberta, Canada.
[27] www.digitalmars.com/d, Jan 2013.

Authors

Mr. Nitesh Chouhan is Head and Assistant Professor in Department of IT at MLV Government Textile & Engineering College, Bhilwara (Rajasthan). He has completed M.E. and B.E. in Computer Science & Engineering. He is having 10 years of academic experience. His research interests are Distributed Computing and Information System Security. E-mail: niteshchouhan_9@yahoo.com

Dr. Maitreyee Dutta is Head and Associate professor in Computer Science & Engineering Department at National Institute of Technical Teachers Training and Research (NITTTR), Chandigarh. She has rich cross-functional experience in continuously delivering in the capacity of teacher and researcher. She has Hands on experience in guiding M.E. and PhD students and producing excellent results. Her research interests are Software Testing and Image processing. E-Mail: dr_maitreyee@gmail.com

Dr. Mayank Singh is Associate professor in Computer Science & Engineering Department at Krishna Engineering College, Ghaziabad. He has good experience in continuously delivering in the capacity of teacher and researcher. He has experience in guiding M.E. and PhD students and producing excellent results. His research interests are Software Testing and Wireless Networks. E-Mail:mayanksingh2005@gmail.com