Application of Hierarchical Slicing to Regression Test Selection of Java Programs

Subhrakanta Panda, Durga Prasad Mohapatra, Ph.D

Selection of regression test case for object-oriented programs is a challenging task. Hierarchical slicing of object-oriented programs helps better in selecting regression test case.

In this paper, first we propose a new slicing method to decompose a Java program into packages, classes, methods and statements that are affected due to the modification in the program. The decomposition is based on the hierarchical characteristic of Java. Then, by mapping these decompositions with the existing test suite, we derive a new test suite and add some new test cases, if necessary, to retest the modified program. We have proposed an intermediate representation of the Java program by considering all the possible dependencies among the program parts. This intermediate representation is used to identify the program constructs that are possibly affected by the change in the program. The packages, classes, methods, and statements thus affected are identified by traversing the intermediate graph, first in the forward direction and then in the backward direction. The test cases covering these affected parts of the program are then selected to retest the program.

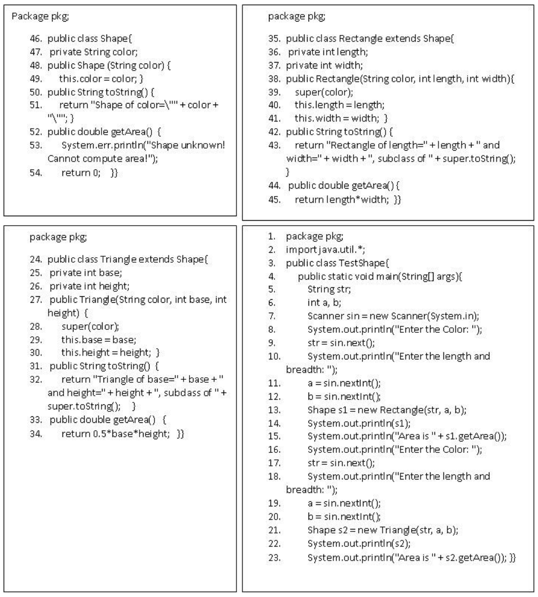
INTRODUCTION

In the software life cycle, regression testing is considered to be an important part. This is because it is essential to validate the modification and to ensure that no other parts of the program have been affected by the change. Regression testing is thus defined as the selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements [1]. A system is said to regress if 1) a new component is added, or 2) a modification done to the existing component affects other parts of the program. Therefore, it is essential to retest not only the changed code but also to retest the possible affected code due to the change. Regression testing is an expensive activity and typically accounts for half of the total cost of software maintenance [2]. It is essential to cut-down on the cost of retesting of the software by following a selective approach to identify and retest only those parts of the program that are affected by the change. In [3] Gupta et al. have identified two important problems in selective regression testing: (1) identifying those existing tests that must be rerun since they may exhibit different behavior in the changed program and (2) identifying those program components that must be retested to satisfy some coverage criterion. The above mentioned problems cover the following important sub-problems associated with regression testing: regression test selection problem, coverage identification problem, test suit execution problem, test suit maintenance problem. Our focus in this paper is to minimize the existing tests so that we can achieve the same coverage thereby reducing the cost and time of retesting of the modified as well as affected parts of the program. So we basically focus on the regression test selection problem in our paper.

In traditional procedure-oriented programs, the approach for regression testing was based on the data flows and control flows within a procedure or among a group of procedures, which was computed by graph reachability algorithms [4, 5] or two-phase graph reachability algorithms [6]. This was mainly achieved by slicing the program dependence graphs (PDG) by using the above algorithms to obtain the sliced program. However, while applying the same techniques to object-oriented programs, we fail because of the presence of many other dependencies originating from the object-oriented features. All though object-oriented features have improved program understandability and readability but have complicated the maintenance activities. The dependencies that arise due to the class and object concept are inheritance dependency, message dependency, data dependency, type dependency, reference dependency, concurrency dependency, etc. These dependencies are represented as edges in our intermediate graph. These dependencies (edges) are formally defined in Section 2.4.

The existing slicing techniques based on system dependence graphs in [7, 8, 9, 10] have considered C++ programs which are partially object-oriented in nature. That's why we are motivated to consider Java programs for our work as it is considered to be a true object-oriented programming language. But the existing techniques cannot be applied to Java programs because of the presence of many new features that increases the dependencies among the program components. The presence of the features like packages, super, dynamic method dispatch, interface, exception handling, multi-threading, etc, in Java adds to the list of dependencies and thus makes the maintenance even more difficult. Their effects on the maintenance of the programs need to be considered separately. In Java, all the classes and their methods are grouped into packages. Suppose a method M1 of class C1 belonging to a package P1 wants to invoke a method M2 of class C2 that belongs to another package P2. This can be achieved by importing the package P2 in package P1 and by instantiating the class C2 in C1. This will create a dependency among the packages P1 and P2, classes C1 and C2, methods M1 and M2 and among the statements in both the methods. Apart from this, there are many methods which are dependent on the type of data they are operating upon. For each type of data, there is a different function. Therefore, using the existing techniques of slicing to slice the SDG of Java programs does not seem to be feasible for regression testing. Incremental regression testing [11] is a probable solution which is based on the following simple observations: (1) if a statement is not executed under a test case, it cannot affect the program output for that test case. (2) Not all statements in the program are executed under all test cases. (3) Even if a statement is executed under a test case, it does not necessarily affect the program output for that test case. (4) Every statement does not necessarily affect every part of the program output. We can apply the above assumptions to Java programs at different levels of packages, classes, methods and statements.

The main objective of this paper is to construct a SDG for Java programs by considering the different dependencies and apply hierarchical slicing [12] to select test cases. We have named our SDG as Extended Object-



```
Package pkg;

46. public class Shape{
47.  private String color;
48. public Shape (String color) {
49.     this.color = color; }
50. public String toString() {
51.     return "Shape of color=\"" + color +
    "\""; }
52. public double getArea() {
53.     System.err.println("Shape unknown!
    Cannot compute area!");
54.     return 0;   }}
```

```
package pkg;

35. public class Rectangle extends Shape{
36.  private int length;
37. private int width;
38. public Rectangle(String color, int length, int width){
39.    super(color);
40.    this.length = length;
41.    this.width = width;  }
42. public String toString() {
43.    return "Rectangle of length=" + length + " and
    width=" + width + ", subclass of " + super.toString();
    }
44.  public double getArea() {
45.    return length*width;  }}
```

```
package pkg;

24. public class Triangle extends Shape{
25.  private int base;
26.  private int height;
27.  public Triangle(String color, int base, int
    height) {
28.     super(color);
29.     this.base = base;
30.     this.height = height;  }
31.  public String toString()  {
32.     return "Triangle of base=" + base + "
    and height=" + height + ", subclass of " +
    super.toString();    }
33.  public double getArea()  {
34.     return 0.5*base*height;  }}
```

```
1.  package pkg;
2.  import java.util.*;
3.  public class TestShape{
4.      public static void main(String[] args){
5.          String str;
6.          int a, b;
7.          Scanner sin = new Scanner(System.in);
8.          System.out.println("Enter the Color: ");
9.          str = sin.next();
10.         System.out.println("Enter the length and
    breadth: ");
11.         a = sin.nextInt();
12.         b = sin.nextInt();
13.         Shape s1 = new Rectangle(str, a, b);
14.         System.out.println(s1);
15.         System.out.println("Area is " + s1.getArea());
16.         System.out.println("Enter the Color: ");
17.         str = sin.next();
18.         System.out.println("Enter the length and
    breadth: ");
19.         a = sin.nextInt();
20.         b = sin.nextInt();
21.         Shape s2 = new Triangle(str, a, b);
22.         System.out.println(s2);
23.         System.out.println("Area is " + s2.getArea()); }}
```

Figure 1 An example Java program

Oriented System Dependency Graph for Java (EOSDGJ). From the point of modification, which is the point of interest for slicing, we compute a forward slice and then a backward slice to determine a set of program components that are affected by the change and all those components which had any impact upon the change. Then taking a hierarchical slice of packages, classes, methods and statements, we identified the impact of change at different programming levels. From the test case coverage analysis, we then selected those test cases that affected at package level, class level, method level and statement level. The outcome of our approach is a set of hierarchically selected test cases that is based upon the proposed EOSDGJ.

The rest of the paper is organized as follows: Section 2 gives a background of program slicing and other related aspects. In Section 3, we discuss our proposed work that is based on the Extended Object-Oriented System Dependency Graph along with a working example. In Section 4, we discuss the implementation of our work. In Section 5, the contribution of many other researchers related to our work is discussed and we have compared our work with some other related work. Section 6 concludes the paper and specifies some work to be carried out in future.

BASIC CONCEPTS

In this section, we discuss the basic concepts and terminologies that are associated with our work and required for understanding our proposed approach.

Program Slicing

Program slicing is a method of separating out the relevant parts of a program with respect to a particular computation. Thus, program slice is a set of statements which affect the value of a variable at a particular point of interest. Program slicing was originally introduced by Mark Weiser [13] as "a method for automatically decomposing programs by analyzing their data flow and control flow starting from a subset of a program's behavior, slicing reduces that program to a minimal form that still produces that behavior". The input that the slicing algorithm takes is usually an intermediate representation of the program under consideration [14]. The first step in slicing a program involves in specifying a point of interest which is called the slicing criterion and is expressed as (s, v), where s is the statement number and v is the variable that is being used or defined at s.

Types of Program Slicing

a. *Forward Slicing:* It comprises of all those parts that might be affected by the slicing criterion because of their dependency on the slicing criterion.
b. *Backward Slicing:* It comprises of all those parts that might affect the slicing criterion because of the dependencies of the slicing criterion on those parts.
c. *Static Slicing:* It comprises of those statements that we get by statically analyzing the code that is by examining some representation of the code without actually executing the program under consideration.
d. *Dynamic Slicing:* It comprises of all those parts of the program that we obtain by actually executing the program with a specific input (included in the slicing criterion). Thus, a dynamic slice is only correct for a specific input whereas a static slice is correct for all inputs.

Over time many researchers have come up with many other types of slicing techniques that can be found in [15, 16, 17].

There are various aspects to be considered in slicing a program. They are listed as follows:

a. *Slicing variable:* Slicing variable may be based on the variables specified in the criteria (slicing point of interest) or it may be on all the variables.
b. *Slicing point:* Considering the slicing point, a programmer's interest may be in observing the impact before or after a particular statement [15].
c. *Slicing direction:* The expected slice of the program may be either in forward direction or backward direction.
d. *Abstraction level:* Abstraction level is either in statement or in procedure level. But considering the typical features of the object-oriented programs, it needs to be extended to class or package level, taking into account the dependencies induced by them.

Hierarchical Slicing

Instead of analyzing the data flow and control flow for the program as a whole, it is preferable to employ the hierarchical structure of the object-oriented programs especially Java programs to detect the impact of the change. A Java program P, is composed of a set of packages, classes, methods and statements. Therefore, in hierarchical slicing, we first try to slice out the packages that might have been affected by the change. From the set of affected packages, we then slice out the affected classes. Then the affected methods and the statements inside those methods are sliced out for retesting. The above concept of hierarchical slicing can be explained by considering a slicing criterion (i.e. point of modification) < s, v >, where s is the statement containing variable v. Let S (P) be the set of identified packages, classes, methods and statements that are affected by the modification to the program. The steps of hierarchical slicing are as follows:

i. We detect the package p containing s and v and all other packages, based on their direct or indirect dependencies on p caused due to import statements. All those packages which are not related to the package p are deleted. Finally, we obtain the package level slice marked as S_l(P).
ii. Then, we analyze S(P), to find out all those classes that are related to the class

containing s and v. All other classes are removed from the slice. The class level slice obtained is marked as $S_2(P)$.

iii. We analyze S(P) and delete all the member methods and variables that are not related to the method containing s and v. The method level slice is marked as $S_3(P)$.

iv. Finally to find out the statement level slice, we analyze S(P) and delete all the statements and predicates that are not related to the statement S containing variable v. The slice obtained is marked as $S_4(P)$.

This stepwise extraction of the slices is known as hierarchical slicing. The test cases obtained at each level can be related as $T(S_4(P)) \subseteq T(S_3(P))$, $T(S_3(P)) \subseteq T(S_2(P))$, $T(S_2(P)) \subseteq T(S_1(P))$. At each level, we obtain more accuracy in minimizing the required number of test cases from a higher level to a lower level by discarding the test cases that are not relevant.

Regression Testing

Testing is an important phase in the software life cycle. It is carried out with the intension of detecting errors in order to improve the quality of the software and to win the confidence of the customer. This phase incurs 60% of the total cost of the software. Therefore, it becomes highly essential to devise proper testing techniques in order to design the test cases so that the software can be tested properly. Testing strategies are based on verification and validation. The static techniques available for testing maps to the verification process without executing the code, whereas the dynamic testing techniques maps to the validation process by executing the code. Regression testing considered as the part of the validation activity possesses a big problem in testing the software. It becomes a big challenge to manage the retesting process with respect to the time and cost, especially when the test suite becomes too large. Therefore, selective retest technique attempts to identify those test cases that can exercise the modified parts of the program and the parts that are affected by the modification to reduce the cost of testing. The features of the selective retest technique are as follows:

a. The resources required to retest a modified version of the program are minimized.

b. This is achieved by minimizing the number of test cases to be exercised.

c. The test suite grows uncontrollably due to the continuous modifications done to the programs for which selective retesting is required.

d. The relationship between the test cases and the program parts that are covered by the test cases can be analyzed better.

Dependency Analysis of the Intermediate Graph

We propose an intermediate representation for Java programs called Extended Object-Oriented System Dependence Graph for Java (EOSDGJ). While constructing EOSDGJ, we have considered some additional dependencies in Java, in addition to the dependencies defined by Krishnaswamy [7] for object-oriented programs. The proposed graph is a set of nodes and edges, where nodes represent the numbered statements and edges represent the different types of dependencies that can exist between the nodes. Some of these dependencies (edges) are identified and defined in [7]. Below, we represent these dependencies (edges) for more clarity and understanding.

i. *Inheritance edge*: The inheritance hierarchy is an important feature of the object-oriented paradigm. It establishes the association between the base class and derived class, in the direction of hierarchy.

ii. *Class membership edge:* Every method in the object-oriented paradigm is a

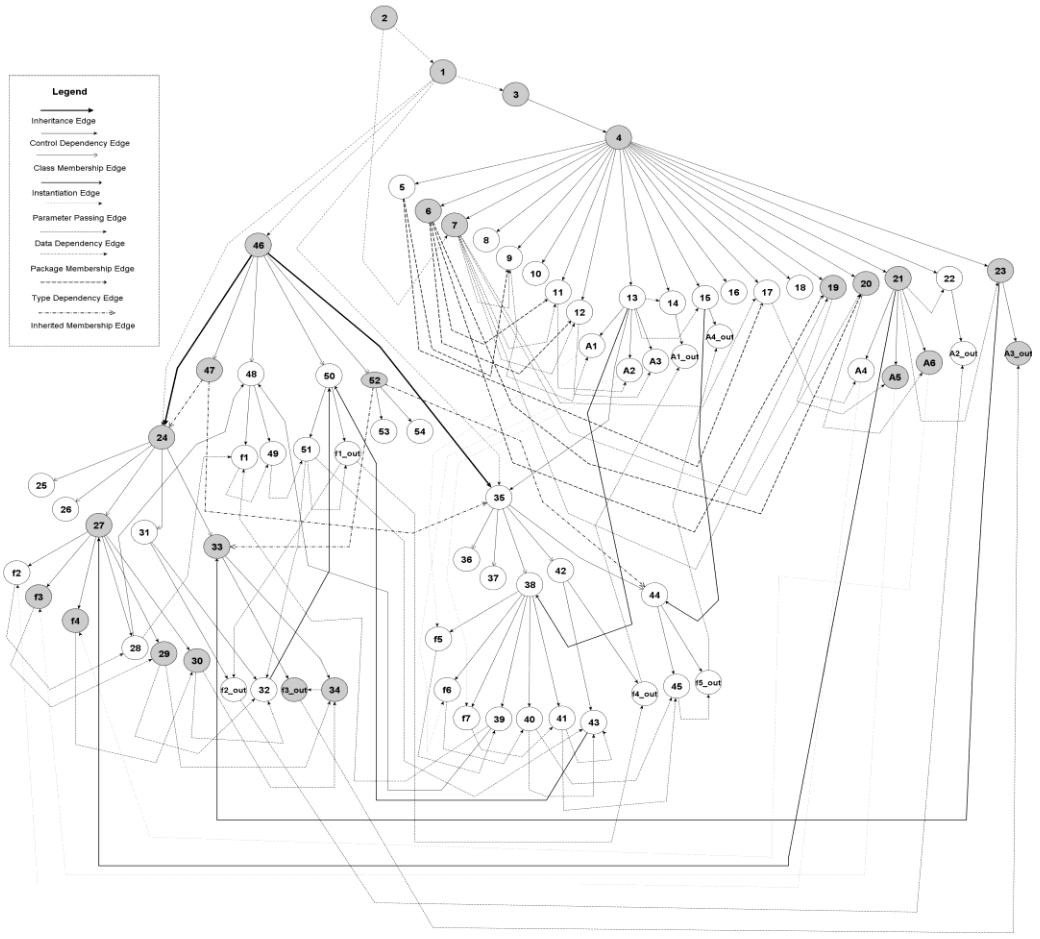


Figure 2 EOSDGJ of the example program

member of a class and is addressable by the object of that class only. Thus, a class membership edge connects the method header and the class header of the class in which the method is defined.

iii. *Inherited membership edge*: Every method and the data members are said to be inherited (irrespective of the access specifier) if they are accessible by the instance of the derived class. Thus, an inherited method or a data member can be considered as an implied member of the derived class. The edge connects the header of the method or data member with the header of the derived class.

iv. *Instantiation edge*: Instantiation means creating the instance of a class by invoking the constructor of the class which initializes the object. The instantiation edge connects the instantiation statement with the class header.

v. *Polymorphic call edge*: A polymorphic edge connects the call statement with the method that is called by resolving the binding dynamically.

vi. *Parameter passing edge*: The parameter passing edge represents the data exchange taking place between the actual parameter and formal parameter vertices.

vii. *Data dependency edge*: When data computed at one statement is used at another statement, an edge is marked to represent the flow of data from the site of computation to the site of usage.

viii. *Control dependency edge*: When the execution of one statement is dependent on the execution of another statement then the former is said to be control dependent on the later. The edge from one vertex to another depicts the control dependence between the vertices in the representation.

Besides these, we have identified the following dependencies (edges) for Java programs.

ix. *Package membership edge*: In Java, all the library classes and user defined classes belong to some package. We have considered the packages as separate nodes in our proposed intermediate representation EOSDGJ. The package dependency arises when one package imports some other packages into it so that some or all the classes in the imported package can be made accessible by instantiating those classes. This creates a dependency between the packages. Thus, an edge from the header of the importing package to the header of the imported package depicts this dependency.

x. *Type dependency edge*: In Java, there are several methods that depend upon the type of data. If the type of data is changed then the method also changes accordingly. Therefore, an edge from the data declaration statement to the statement containing a call to such a method is essential to depict the type dependency.

PROPOSED WORK

In this section, we propose an algorithm, which we named *Hierarchical Regression Test Selection (HRTS)* algorithm for generating selective regression test cases. We maintain all the test cases along with the information of their coverage of packages, classes, methods and statements in Table 1. In our proposed work, we also maintain the following sets of information:

$P = \{p_1, p_2 \ldots p_n\}$ is the set of all the packages that are used in the given program.

$C = \{c_1, c_2 \ldots c_n\}$ is the set of all the classes defined in the program.

$M = \{m_1, m_2 \ldots m_n\}$ is the set of all the methods defined in the program.

$S = \{s_1, s_2 \ldots s_n\}$ is the set of all the statements in the program.

Notations Used:

i. *Q* – Queue that contains all the nodes reached in the forward traversal of the EOSDGJ graph.

ii. *U* - The set containing all the packages, classes, methods and statements that are affected by the modification and that are executed by the test cases of the program.

iii. *Pk* - The set of packages extracted from EOSDGJ that are affected by the modification.

iv. *Cl* - The set of classes extracted from EOSDGJ that are affected by the modification.

v. *Mt* - The set of methods extracted from EOSDGJ that are affected by the modification.

vi. *St* - The set of statements extracted from EOSDGJ that are affected by the modification.

Now, we describe our proposed HRTS Algorithm:

Algorithm HRTS

Step 1: Construct the EOSDGJ for the program.

Step 2: Do the following:

i. Initialize *Q*, *U*, *Pk*, *Cl*, *Mt*, *St* to NULL.

ii. Traverse the proposed EOSDGJ using the Depth First Search (DFS) algorithm. First traverse in the forward direction, starting from the point of modification (slicing criterion). Then, detect all those program parts (nodes) that are dependent on the modified statement and hence might be affected by the modification.

Step 3: Add each node of the graph that is reached by the traversal algorithm to a queue, *Q*.

Step 4: For each node $v \in Q$, do the followings:

i. Remove *v* from *Q* and add it to set *U*.

ii. Taking *v* as the starting point, we traverse backward using DFS algorithm to extract all those nodes on which node *v* is dependent on and add them to set *U*.

iii. Repeat Step 3, till *Q* is empty.

Finally, the set *U* will contain all program parts affected by the modification.

Step 5: Do the following computations to obtain the hierarchical slice:

i. $Pk = P \cap U$, if the set *Pk* is non-empty then we get the set of packages that are affected by the modification.

ii. $U = U - Pk$, now set *U* consists of only classes, methods and statements.

iii. $Cl = C \cap U$, if the set *Cl* is non-empty then we get the set of classes that are affected by the modification.

iv. $U = U - Cl$, now set *U* consists of only methods and statements.

v. $Mt = M \cap U$, if the set *Mt* is non-empty then we get the set of affected methods.

vi. $U = U - Mt$, now set *U* consists of only affected statements.

vii. $St = U$

Here *Pk*, *Cl*, *Mt*, *St* are the sliced sets of packages, classes, methods and statements respectively, extracted from the EOSDGJ, which might have been affected by the modification done to the program.

Step 6: Select the test cases step by step from the

Table 1 Test case distribution for the example program in Figure 1

TestCases	*Packages*	*Classes*	*Methods*	*Statements (nodenos.)*
T1 – T5	node1 : pkg	node24 : Triangle	node31 : toString ()	32
			node27 : Triangle ()	25, 26, 28, 29, 30
		node46 : Shape	node50 : toString ()	51
			node48 : Shape ()	47, 49
		node3 : TestShape	node4 : Main ()	5, 6, 7, 16, 18, 19, 20, 21, 22
T6 – T10	node1 : pkg	node24 : Triangle	node33 : getArea ()	34
			node27 : Triangle ()	25, 26, 28, 29, 30
		node46 : Shape	node52 : getArea ()	53, 54
			node48 : Shape ()	47, 49
		node3 : TestShape	node4 : Main ()	5, 6, 7, 16, 18, 19, 20, 21, 23
T11 – T15	node1 : pkg	node35 : Rectangle	node42 : toString ()	43
			node38 : Rectangle ()	36, 37, 39, 40, 41
		node46 : Shape	node50 : toString ()	51
			node48 : Shape ()	47, 49
		node3 : TestShape	node4 : Main ()	5, 6, 7, 8, 9, 10, 11, 12, 13, 14
T16 – T20	node1 : pkg	node35 : Rectangle	node44 : getArea ()	45
			node38 : Rectangle ()	36, 37, 39, 40, 41
		node46 : Shape	node52 : getArea ()	53, 54
			node48 : Shape ()	47, 49
		node3 : TestShape	node4 : Main ()	5, 6, 7, 8, 9, 10, 11, 12, 13, 15

package level to the statement level.

i. Let there be n number of test cases in the test suite T, where $T = \{t_1, t_2 \ldots t_n\}$. The set of packages covered by each test case t_i, $i = 1, 2 \ldots n$ is represented by P_{ti}. Determine the set of test cases selected at the package level for retesting the program.

$T' = \{\}$ \\ T is initialized.

$\forall t_i \epsilon T, P'_t = Pk \cap P_{ti}$

If P'_t is non-empty, then

$T' = T' \cup t_i$, where T' is the set of selected test cases at package level.

ii. Determine the set of test cases selected at the class level

$T'' = \{\}$ \\ T'' is initialized.

$\forall t_i \epsilon T', C'_t = Cl \cap C_{ti}$

where C_{ti} is the set of methods covered by the test case t_i.

If C'_t is non-empty, then

$T'' = T'' \cup t_i$, where $T'' \subseteq T'$ is the set of selected test cases at class level.

iii. Determine the set of test cases selected at the method level

$T''' = \{\}$ \\ T''' is initialized.

$\forall t_i \epsilon T'', M'_t = Mt \cap M_{ti}$

where M_{ti} is the set of methods covered by the test case t_i.

If M'_t is non-empty, then

$T''' = T''' \cup t_i$, where $T''' \subseteq T''$ is the set of selected test cases at method level.

iv. Finally, determine the statement level slice

$T^f = \{\}$ \\ T^f is initialized.

$\forall t_i \epsilon T''', S'_t = St \cap S_{ti}$

where S_{ti} is the set of statements covered by the test case t_i.

If S'_t is non-empty, then

$T^f = T^f \cup t_i$, where $T^f \subseteq T'''$ is the set of selected test cases at statement level.

Complexity Analysis of HRTS Algorithm

The Complexity analysis of our algorithm is as follows:

Let the program under consideration have N statements. Each node in the proposed EOSDGJ represents a single statement of the program. However, some extra nodes are required to represent the actual and formal arguments of method invocation and method definition. For such statements in the program, the number of extra nodes required is equal to the number of actual and formal arguments present in the program. Let us assume that too many parameters in a method definition are not allowed. Let the number of parameters present in the program be k, where k is some bounded small positive number. If each statement of the program is represented in the EOSDGJ by some k number of extra nodes (assuming each statement has actual and formal arguments), then it can be stated that the space requirement of the EOSDGJ is $0(kN^2)$.

Since k is a small bounded positive integer, we can conclude that the space requirement of the EOSDGJ is $0(N^2)$. Apart from this, some additional space is required by the algorithm in maintaining the packages, classes, methods, statements and the coverage information for each test case. The additional space requirement is as follows:

i. We have assumed that the total number of lines of code in our program is N. Therefore, the number of packages, classes, methods and statements present in the program will be less than N. So, we can say that the space required to maintain this additional information about packages, classes, methods and statements present in the program will be $0(N)$.

ii. Let the number of test cases used to test the original program be m, where m is a bounded positive integer. Each test case will maintain the coverage information of packages, classes, methods and statements. Assuming that each test case covers all the packages, classes, methods and statements in the program, the total space requirement would be 0(mN). As m is a bounded positive integer, so the space requirement is 0(N).

Let N be the set of vertices and E be set of edges in EOSDGJ. Since, each node in the graph is visited (using DFS Algorithm) only once, so the time complexity is (N + E). If the time spent in each recursive call is ignored, then each vertex u can be processed in O $(1 + d^{+}_{G}(u))$ time. So the total time required for our algorithm is given by

$$\begin{aligned} \text{Total Time} &= N + \Sigma_{u\epsilon N} (1 + d^{+}_{G}(u)) \\ &= N + \Sigma_{u\epsilon N} d^{+}_{G}(u) + N \\ &= 2N + E \\ &\approx \Theta(N + E) \end{aligned}$$

The operations involved in the algorithm for hierarchical slicing and selection of test cases are intersection and union which require two sets as operands. Assuming that each set contains N elements, the worst-case run time of each of the above operations will be 0 (N^2). Therefore, the worst-case run time of our algorithm is 0 (N^2).

We have used DFS algorithm to traverse the proposed graph instead of BFS algorithm because of the following disadvantages of BFS algorithm:

i. Let us assume that each node in the graph EOSDGJ has a branching factor of b. That means, each node in the graph has b number of child nodes. Since, BFS algorithm requires all the nodes to be stored in a queue for further traversal, at level one, b numbers of nodes are generated. At level 2, b^2 numbers of nodes will be generated and so on. The number will increase in an exponential manner requiring a huge amount of space in the order of 0 (b^n), to store all the generated nodes for further exploration. Assuming the depth of the graph to be n, the total space required will be $b + b^2 + ... + b^n$.

ii. Also the time complexity of processing each of these nodes will increase exponentially.

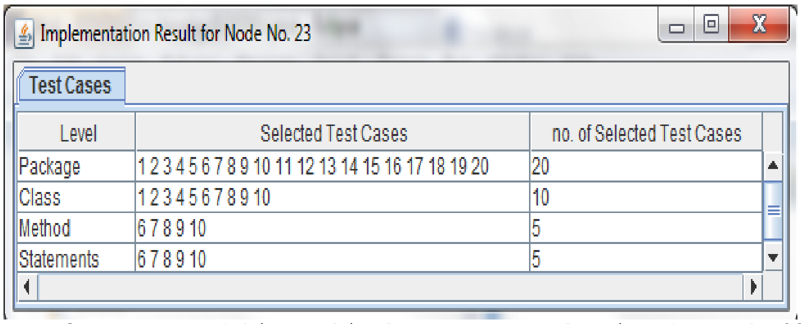
Working of the HRTS Algorithm

Figure 1 contains an example Java program. The corresponding EOSDGJ of the program is shown in Figure 2. Suppose, the object s2 in line 23 of the example program is changed to s1. Therefore, the method of class *Rectangle* in line 44 will be invoked instead of the method of class *Triangle* in line 33. As a result, the output will be erroneous. We first traverse in the forward direction from the point of modification to determine all those nodes/program parts that may be affected by the change. The nodes detected to be dependent on line 23 are *23, 33, 34, A3_out, f3_out*. Then, from each of the selected nodes detected in the forward traversal, we traverse in the backward direction to determine all those nodes that might have affected the selected nodes. The nodes that are finally selected by the backward traversal are shown as shaded nodes in Figure 2. The two package nodes that are present in the slice are *node 1* and *node 2*. So from Table 1, all the test cases T1 - T20 covering the two package nodes are selected. The class nodes that are sliced are *node3, node46* and *node24*.

In the second level, the classes that are selected are *Triangle, Shape, TestShape*. Therefore, test cases T1 - T10 are selected out of the 20 test cases selected at the first level. Similarly, in the third level i.e. the method level the selected test cases are T6 - T10. Since, all these 5 test cases cover the statement level slice for the example program, so at the fourth level all these are selected. Finally, we have the five test cases that can be used to retest the program as shown in Table 2.

Table 2 Summary of test case selection for the example program in Figure 1

Level	*Selected Test Cases*	*Number of Selected Test Cases*
Package	$T1-T20$	20
Class	$T1-T10$	10
Method	$T6-T10$	5
Statement	$T6-T10$	5


Implementation Result for Node No. 23

Test Cases

Level	Selected Test Cases	no. of Selected Test Cases
Package	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20	20
Class	1 2 3 4 5 6 7 8 9 10	10
Method	6 7 8 9 10	5
Statements	6 7 8 9 10	5

Figure 3 Summary of hierarchical test case selection for Node 23

IMPLEMENTATION

In this section, we describe briefly the implementation of our work. We have considered five programs for our experiment, out of which three are simple programs and the other two are little complex programs developed by us. The modifications that were made to the above mentioned programs include modification to the data types of member variables, modification of expressions in a method, modification of the object relation, addition of a new member variable and deletion of a new member variable, etc. The results of our experiment are given in Table 3. In Figure 3, we show the implementation result of hierarchical test case selection for the input node 23.

COMPARISION WITH RELATED WORK

In this section, we discuss the work related to our work and then compare some of the approaches with our approach. First, we discuss the available related work on program slicing. Then, we discuss the existing related work on regression testing.

Table 3 Result obtained for Regression testing of different programs

Program	*Lines of Code*	*Test Cases*	*Selected Test Cases*
P1	54	20	5
P2	75	15	7
P3	90	25	10
P4	300	30	8
P5	900	50	12

Program Slicing

Many researchers have proposed several methods for program slicing [4, 6, 10, 13, 18, 19]. Some of the recent applications of program slicing are described in [31, 32]. In [7], Krishnaswamy augmented the SDG with some more dependencies relevant to object-oriented programs. But, these dependencies do not completely cover a true object-oriented program such as a Java program. In our proposed EOSDGJ, we have added some new dependencies applicable to Java programs, such as package and type dependencies. This covers a true object-oriented program. Harrold et al. [20] proposed an algorithm to identify the dangerous edges for safe regression test selection. This method compared two nodes in the proposed Java Interclass Graph (JIG) of P and P′ to identify the execution path of a test case in P and P′, so that it can be known whether any edge is dangerous or not. To make the comparison between the nodes they have used the lexicography equivalence of the text labeled on each node. For example, if a class Y in package pkg extends a class X in the same package, and X implements interface I in package abc, then the text associated with the node for class Y will be Java.lang.Object:abc.I:pkg.X:pkg.Y. As the level of inheritance will be deeper, the text will become lengthier and comparison will incur more runtime overhead. In our approach, we do not require to do any such comparisons. So, we save time by avoiding this computational overhead.

Many researchers have proposed different approaches to compute slices for Java programs. Some of the slicing mechanisms are based on the dependency graphs like PDG and SDG [21, 22], while other approaches are based on the Java bytecode analysis [23, 24].

In [21 - 24], the mechanism to slice a Java program is based on specific feature or type of dependency present in the program under consideration. Whereas, the overall impact of the features on the dependency such as the dependency due to the presence of packages and other specific Java features is not considered. Our approach has made a decent effort in analyzing all the possible dependencies and computing a more accurate slice. To be able to employ slicing in regression testing, it is necessary to identify all those statements that affect the modified statement and those statements that may get affected by the modification. But, most of the existing approaches [21, 22] are based upon forward traversing or backward traversing. This will only result in the partial identification of the affected statements due to the modification. But, our approach gives a better result for regression testing. Both forward and backward traversal of our approach correctly finds all the program parts that get affected and that may affect other program parts due to the change.

Regression Testing

Software maintenance being the most important and expensive activity in the process of Software Development Life Cycle (SDLC), many researchers have proposed several approaches for ordering the test cases of procedural programs. Rothermel [25, 26] and Elbaum [27] have considered different types of program coverage criteria such as total statement coverage, additional statement coverage, total function coverage etc. Jeffrey and Gupta [28], proposed a method for prioritizing the test cases for regression testing based on the coverage of relevant slice of the output of a test case. They assigned test case weights to the test cases to determine their priority. They determined the test case weight by summing up the number of statements present in the relevant slice and number of statements exercised by the test case. Korel et al. [29] prioritized the regression test suite by considering the state model of the system. Whenever, the source code was modified, the corresponding change in its state model was identified. These modified transitions along with the runtime information were used to prioritize the test cases. However, the available techniques were of little help when they were applied to regression testing of object-oriented programs.

Harrold et al. [20] have proposed traversal algorithms to identify the dangerous edges for safe regression test selection. The dangerous edge is defined to be an edge e such that for each input i causing P to cover e, P(i) and P′(i) may behave differently due to differences between P and P′, where P and P′ are the programs under consideration and the modified program respectively. The dangerous edge is identified by traversing the proposed Java

Interclass Graph (JIG). This method compared two nodes of P and P′ in the JIG to identify the execution path of a test case in P and P′, so that it can be known whether any edge is dangerous or not. This technique ensures that any test case that does not cover the dangerous entity will behave in the same way in both P and P′. Thus, it cannot expose new faults in P′. So, it is safe to select only those test cases for which the dangerous entity is covered.

Li et al. [30] used hierarchical slicing for regression test case selection. Their proposed model consisted of three levels: syntax analysis, generation of dependence graphs, and computation of slices. They proposed different dependence graphs such as package level dependence graph (PLDG), class level dependence graph (CLDG), method level dependence graph (MLDG) and statement level dependence graph (SLDG) were based on the slicing criteria. When any modification is done to a statement, the dependency of that statement with its method, class and package can be easily detected because of the different levels of graphs maintained. Identification of other packages, classes, methods and statements related to the modified statement can also be easily done. The overall performance had improved as the irrelevant packages, classes, methods and statements were discarded from the generated graph. But, the proposed method required all the different graphs (PLDG, CLDG, MLDG, SLDG) to be generated for each change done to the program and was not very advantageous in case of frequent changes. Thus, to avoid the above mentioned problem, the slicing criterion was fixed. We have implemented the hierarchical slicing technique on the EOSDGJ which does not depend on any fixed change. It rather works for any number of changes done to any statement, without requiring us to maintain additional graphs. If the change made to the example program triggers some new change to be made, then our approach is capable to handle it.

Tao et al. [12] applied hierarchical slicing for regression testing of object-oriented programs. In their approach, they have also proposed to maintain separate graphs for packages, classes, methods and statements even if they were not affected by the change. This again needs more space requirement. This is because with the increase in the program complexity, there will be an increase in the number of packages, classes, methods and statements which are required to be represented as separate graphs. But, in our approach, we only maintain the EOSDGJ graph. This does not impose any additional space requirement. In some papers [25, 27, 28] only control dependency and data dependency are considered. We have identified some more dependencies such as package membership dependency and type dependency and have represented various object relations so as to consider more features of Java programs and computed the slices more accurately. So, the appropriate test cases for regression testing will be selected more accurately.

CONCLUSION AND FUTURE WORK

We have proposed an application of slicing to regression test selection based on the Extended Object-Oriented System dependency Graph for Java programs (EOSDGJ). We have considered some new dependencies in addition to control and data dependencies that play a crucial role in regression test selection. The selected test cases were found to be very efficient in detecting the regression errors. In our future work, we will focus on reducing the space requirement of our algorithm. We will also explore the application of other variants of slicing in regression test selection for more complex Java programs. We will also try to use slicing techniques for prioritizing the test cases both for object-oriented as well as aspect-oriented programs.

REFERENCES

1. N Chauhan. Software Testing Principles and Practices, chapter 8. Oxford University Press, New Delhi, India, pages: 255-273, 2010.
2. H Leung and L White. Insights into Regression Testing Selection. In Proceedings of the Conference on Software Maintenance, pages: 60–69, 1989.
3. R Gupta, M J Harrold, and M L Soffa. Program Slicing-Based Regression Testing Techniques. Software Testing, Verification and Reliability, Vol. 6, No. 2, pages: 83–111, 1996.
4. S Horowitz, T Reps, and D Binkley. Interprocedural Slicing using Dependence Graphs. ACM SIGPLAN Notices, Vol. 23, No. 7, pages: 35–46, 1988.
5. S Horowitz and T Reps. The use of Program Dependence Graphs in Software Engineering. In Fourteenth International Conference on Software Engineering, Melbourne, pages: 392–411, 1992.
6. J K Ottenstein and M L Ottenstein. The Program Dependence Graph in a Software Development Environment. ACM SIGPLAN Notices, Vol. 19, No. 5, pages: 177–184, 1984.
7. A Krishnaswamy. Program Slicing: An Application of Object-Oriented Program Dependence Graphs. Technical Report TR94-108, Department of Computer Science, Clemson University, 1994.
8. D P Mohapatra, R Mall, and R Kumar. An Overview of Slicing Techniques for Object-Oriented Programs. Informatica (Slovenia), Vol. 30, No. 2, pages: 253–277, 2006.
9. J Zhao. Dynamic Slicing of Object-Oriented Programs. Technical Report SE-98-119, Information Processing Society of Japan, 1998.
10. L Larsen and M J Harrold. Slicing Object-Oriented Software. In Proceedings of the 18th IEEE International Conference on Software Engineering, pages: 495–505, 1996.
11. I Forgacs and A Bertolino. Feasible Test Path Selection by Principal Slicing. In Proceeding of 6th European Software Engineering Conference, 1997.
12. C Tao, B Li, X Sun, and C Zhang. An Approach to Regression Test Selection Based on Hierarchical Slicing Technique. In 34th Annual IEEE Computer Software and Applications Conference Workshops, pages: 347–352, 2010.
13. M Weiser. Program Slicing. In Proceedings of the 5th International Conference on Software, San Diego, California, USA, pages: 439–449, 1981.
14. J Zhao, J Cheng, and K Ushijima. A Dependence Based Representation for Concurrent Object-Oriented Software Maintenance. In Proceedings of 2nd Euromicro Conference on Software Maintenance and Reengineering, pages: 60–66, 1998.
15. G A Venkatesh. The Semantic Approach to Program Slicing. ACM SIGPLAN Notices, Vol. 26, No. 6, pages: 107–119, 1991.
16. G Canfora, A Cimitile, and A D Lucia. Conditioned Program Slicing. Information and Software Technology, Vol. 40, pages: 595–607, 1998.
17. M Harman, D Binkley, and S Danicic. Amorphous Program Slicing. The Journal of Systems and Software, Vol. 68, pages: 45–64, 2003.
18. B Korel and J Laski. Dynamic Program Slicing. Information Processing Letter, Vol. 29, No. 3, pages: 155–163, 1988.
19. H Agrawal and J Horogan. Dynamic Program Slicing. In Proceeding of ACM SIGPLAN'90 Conference on Programming Language Design and Implementation,

SIGPLAN Notices, Analysis and Verification, pages: 246–256, 1990.

20. M J Harrold and et al. Regression Test Selection for Java Software. In Proceeding of the ACM Conference on OO Programming, Systems, Languages, and Applications (OOPSLA'01), pages: 312–326, 2001.

21. Z Chen and B Xu. Slicing Object-Oriented Java Programs. ACM SIGPLAN Notices, Vol. 36, No. 4, pages: 33–40, April 2001.

22. M Allen and S Horwitz. Slicing Java Programs that Throw and Catch Exceptions. In ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'03), ACM, pages: 44–54, June 2003.

23. T Wang and A Roychoudhury. Using Compressed Bytecode Traces for Slicing Java Programs. In 26th International Conference on Software Engineering (ICSE'04), ACM, pages: 512-521, 2004.

24. C Hammer and G Snelting. An Improved Slicer for Java. In Workshop on Program Analysis for Software Tools and Engineering (PASTE'04), 5th ACM SIGPLAN-SIGSOFT, pages: 17–22. ACM, 2004.

25. G Rothermel, R Untch, C Chu, and M Harrold. Prioritizing test cases for Regression Testing. IEEE Transactions on Software Engineering, Vol. 27, No. 10, pages: 924–948, 2001.

26. A Malishevsky, J Ruthruff, G Rothermel, and S Elbaum. Cost-cognizant Test Case Prioritization. Technical Report TRUNL-CSE-2006-0004, 2006.

27. S Elbaum, A Malishevsky, and G Rothermel. Test Case Prioritization: A family of Emprical Studies. IEEE Transactions of Software Engineering, Vol. 28, No. 2, pages: 159–182, 2002.

28. D Jeffrey and N Gupta. Test Case Prioritization using Relevant Slices. In Proceedings of 30th Annual International Computer Software and Applications Conference, pages: 411–420, 2006.

29. B Korel, G Koutsogiannakis, and L Tahat. Application of System Models in Regression Test Suite Prioritization. In Proceedings of the IEEE International Conference on Software Maintenance, pages: 247–256, 2008.

30. Bi Xin Li, Xiao Cong Fan, Jun Pang, and Jian Jun Zhao. Model for Slicing Java Programs Hierarchically. Journal of Computer Science and Technology, Vol. 19, No. 6, pages: 848–858, 2004.

31. W Wen. Software Fault Localization based on Program Slicing Spectrum. In the Proceedings of the 2012 International Conference on Software Engineering, ACM, pages: 1511-1514, 2012.

32. D Wang, M Dong and W Zhan. An Input Data Related Behavior Extracting and Measuring Model. International Journal of Applied Mathematics & Information Sciences, Vol. 7, No. 2, pages: 683-689, 2013.

Author Profile

Subhrakanta Panda is a Ph.D scholar in the Department of Computer Science & Engineering, National Institute of Technology (NIT), Rourkela, India. He can be contacted at 511cs109@nitrkl.ac.in.

Durga Prasad Mohapatra, Ph.D is currently working as an Associate Professor in the Department of Computer Science and Engineering, National Institute of Technology (NIT), Rourkela, India. His research interests include software engineering, real-time systems, and discrete mathematics and distributed computing. Dr. Mohapatra has co-authored the book Elements of Discrete Mathematics: A computer Oriented Approach published by Tata Mc-Graw Hill. He can be contacted at durga@nitrkl.ac.in.