CSIT (June 2013) 1(2):159–173
DOI 10.1007/s40012-013-0011-7

ORIGINAL RESEARCH

An approach to prioritize the regression test cases of object-oriented programs

Chhabi Rani Panigrahi · Rajib Mall

Received: 12 August 2012 / Accepted: 14 March 2013 / Published online: 10 April 2013

Abstract We propose a regression test case prioritization technique for object-oriented programs. We first construct an intermediate graph model of a program from its source code. When the program is modified, the model is updated to reflect the changes. Our constructed model represents control and data dependencies, and information pertaining to various types of dependencies arising from object-relations such as association, inheritance and aggregation. We determine the affected nodes in the model by constructing the union of the forward slices corresponding to each changed model element. A test case that covers a larger number of affected model elements is given higher priority. Our experimental results indicate that our approach on an average achieves an increase in the APFD metric value by 25.70 % as compared to a related approach.

Keywords Software maintenance · Regression testing · Regression test case prioritization · Slicing

1 Introduction

Regression testing involves rerunning relevant test cases from existing test suite to build confidence that there is no unintended side-effects due to changes made to a software. A naive approach would be to run the entire test suite after every change. This is usually costly, wasteful of resources, and requires unduly long time. To overcome this problem a large number of regression test selection techniques have been proposed [1–3]. However, regression test selection techniques often select significant number of test cases even for small changes made to a program. Researchers have attempted to make regression testing more effective by developing various approaches such as test suite minimization (TSM) [4–7] and test case prioritization (TCP) [8, 9, 10, 11]. These techniques usually target to reduce the cost or improve the effectiveness of regression testing.

Regression TCP techniques provide a method to order test cases according to certain criteria, so that the highest priority test cases can be executed earlier in a regression testing session. Further, when it is possible to execute only a few test cases, the most fault-revealing test cases can be executed. TCP techniques usually target to order test cases based on the rate of fault detection or the rate of code coverage [8, 9]. Most of the TCP techniques proposed in the literature were developed in the context of procedural programs. These existing TCP techniques do not work satisfactorily [8–10] for object-oriented programs, because these do not consider the implicit dependencies that arise due to object-relations.

Binder pointed out that in object-oriented programs methods tend to be small and simple [12]. Therefore, the programming complexities move from intra-procedural to inter-procedural aspects. Therefore, it can be assumed that consideration of dependencies arising on account of object-relations in regression test prioritization is important. In this context, we have used a program model that represents the object-relations in an object-oriented program in addition to the traditional program dependencies. Again a test case which covers higher number of affected nodes in the dependency model has a higher chance to detect an error(s) than a test case which covers less number of affected nodes. Based on the above fact, we propose a

C. R. Panigrahi (✉) · R. Mall
Department of Computer Science & Engineering, Indian Institute of Technology Kharagpur, Kharagpur, India
e-mail: panigrahichhabi@gmail.com; chhabi@cse.iitkgp.ernet.in

R. Mall
e-mail: rajib@cse.iitkgp.ernet.in

regression TCP technique that prioritizes test cases according to the number of affected nodes covered by a test case in the dependency model of an object-oriented program.

This paper is organized as follows: In Sect. 2, we discuss certain background concepts used in our work. Review of the related regression TCP techniques is presented in Sect. 3 and our regression TCP technique is described in Sect. 4 In Sect. 5, we describe the experimental study carried out by us to prove the effectiveness of our approach. In Sect. 6, we present a comparison of our technique with related work and finally conclude the paper in Sect. 7.

2 Basic concepts

In this section we describe some basic concepts that form the basis of our work.

2.1 SDG of object-oriented programs

System dependence graph (SDG) was first introduced by Horwitz et al. [13] and was used to model procedural programs. Later on, SDG was extended by Larsen and Harrold to model object-oriented programs [14] and is named as extended system dependency graph (ESDG).

An ESDG 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 vertices and edges in an ESDG.

2.1.1 ESDG vertices

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

2.1.2 ESDG edges

– *Data dependence* edge: *Data dependence* edges are used to represent the data dependence relations existing among different *statement* vertices.
– *Control dependence* edge: A *control dependence* edge is used to represent control dependence relations between two *statement* vertices.
– *Call* edge: A *call* edge is used to connect a call site to a *method entry* vertex and also to connect various possible polymorphic method call vertices to a *polymorphic choice* vertex.
– *Parameter dependence* edge: *Parameter dependence* edges are used for passing values between actual and formal parameters in a method call. *Parameter dependence* edges are of two types: *parameter-in* and *parameter-out* edges.
– *Summary* edge: A *summary* edge represents the transitive dependence between *actual-in* and *actual-out* vertices.
– *Class member* edge: A *class member* edge is used to 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.

A class is represented in an ESDG by a class dependence graph (ClDG) [3]. Each method in a ClDG is represented by a procedure dependence graph [15]. The root node of a ClDG is represented by a *class entry* vertex. Each method in a class is associated with 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, method calls are represented by *call* vertices. A *formal-in* vertex is added for each formal parameter in the method, and one *formal-out* vertex for each formal reference parameter that is modified by the called method.

The methods of a class can interact among each other or with methods of other classes. 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 added for each *called* method. The *actual-in* parameter vertices are connected to the corresponding *formal-in* vertices in the 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. A ClDG represents effects of *return* statements by connecting each *return* statement to its corresponding *call* vertex using a *parameter-out* edge. Similarly, for the *actual-in* parameters that may affect the returned value, *summary* edges are added between the *actual-in* vertices and the *actual-out* vertex. For a derived class, the representation of the base class method is reused for representing the inherited methods. Apart from connecting the *class entry* vertex of a class to its *method entry* vertices of locally defined methods, *class member* edges are constructed to connect the *class entry* vertex of the derived class to the *method entry* vertices of the methods inherited by the derived class.

In an object-oriented program, a class may instantiate another class. When a class C_1 instantiates another class C_2, an implicit call to the constructor of C_2 is made. To represent instantiation of a class C_2 by C_1 in ESDG, a *call* vertex is created in C_1. A *call* edge connects this *call* vertex to the constructor of C_2. When a method m_1 in a class C_1 calls a public method m_2 of another class C_2, the *call* vertex in C_1 is connected to the *entry* vertex of m_2 by a *call* edge.

An ESDG can also represent polymorphic relationships between a calling method and its called methods. A polymorphic method call in an object-oriented program implies that the destination of a method call is determined at runtime. An ESDG uses a *polymorphic choice* vertex to represent the dynamic choice among the possible destinations. The *polymorphic choice* vertex has *call* edges that are connected to the subgraphs representing calls to each possible destinations. Figure 2 shows the partial ESDG representation of the class *Coinbox* for the Java code as given in Fig. 1. As shown in Fig. 2, *class entry* node of *Coinbox* class is connected to all the *method entry* nodes by *class member* edges. For readability sake, we have not shown all the edges in the ESDG.

2.2 Regression testing cycle

After a software is released, the failure reports and change requests from the user are accumulated using a defect tracking system [16]. At suitable intervals the program is modified to address all change requests and failure reports. After the required changes have been carried out, *regression* testing is carried out to revalidate the affected parts of the code. The failures identified by the regression testing are fixed and this cycle is repeated until there are no more regression test failures. Each cycle of regression testing and the fixing of identified bugs is referred to as a regression testing cycle.

The activities undertaken during a regression testing cycle include test setup, test selection, test prioritization, test case execution and failure report preparation. After regression testing are successfully completed, a new version of the software is released, which then undergoes similar regression testing cycles. Please note that, there can be multiple regression testing cycles before each release of a software.

2.3 Program slicing

The concept of a program slice was first introduced by Weiser to aid debugging of programs [17]. Since then a significant amount of research results on code slicing have been reported in the literature [18]. Code slicing techniques have been used in various application areas such as program debugging, testing, program understanding, and software maintenance [13, 19]. A program slice consists of all those program statements that can potentially affect the values computed at some point of interest called the slicing criterion [13, 14]. For different applications, different types of program slices have been proposed [18]. A *backward* program slice at a program point p with respect to a variable v contains all statements in the program, including conditionals, that might affect the value of v at p [13], whereas a *forward* program slice at a program point p with respect to a variable v contains all statements in the program, including conditionals, that might be affected by any modifications to v at p [13].

Slice is calculated as a graph-reachability problem in the dependence graph of a program. The inter-procedural slicing algorithm was first proposed by Horwitz et al. [13]. Later on, Horwitz et al.'s algorithm was extended by Larsen and Harrold [14] for slicing on a dependence graph of an object-oriented software. A comprehensive survey on program slicing techniques has been reported in [20].

2.4 APFD metric

To measure the average rate of fault detection of a regression test suite the average percentage of faults detected (APFD) metric was proposed by Rothermel et al. [21]. The APFD metric has been used by several researchers [22, 23] to evaluate the effectiveness of a test prioritization scheme. The APFD metric for a test suite is calculated by taking the weighted average of the number of faults detected during execution of the program with the test suite. APFD metric values range from 0 to 100 and a higher number indicates a faster rate of fault detection.

The APFD metric can be calculated using the following expression. Let T be the original test suite containing n test cases, and let F be a set of m faults revealed by T. Let T' be an ordering of T. In T', let TF_i be the first test case which reveals a fault i. Then the APFD metric for test suite T' can be obtained by using the equation:

$$APFD = 1 - \frac{TF_1 + TF_2 + \ldots\ldots + TF_n}{n * m} - \frac{1}{2n} \tag{1}$$

For example, consider a program with a test suite of five test cases, that have been labeled a through e. Suppose

Fig. 1 Java code of the class *Coinbox*

```
CE1:  class Coinbox {
 E2:   public Coinbox(unsigned flag) {
 S3:       totCoins  = curCoins = alVend = 0 ;
 S4:       checkflag = flag; }
 E5:   public void insert ( ) {
 S6:      curCoins ++ ;
 S7:      if ( curCoins > 1 )
 S8:          alVend = 1 ;
 S9:      if (checkflag)
S10:          check( );
        }
E11:  public void vend (unsigned selection) {
S12:  if (isalvend)  {
S13:     totCoins  + = curCoins ;
S14:     curCoins = 0 ;
S15:    alVend = 1;
S16:        d1. dispense (selection) ;   }
      }
E17:  public unsigned isalvend ( ) {
S18:       return alVend;  }
E19:   public void return ( )  {
S20:       c1. retcoins (curCoins);
S21:       curCoins = 0 ;
S22:      alVend = 0 ;   }
S23:    if (checkflag )
S24:        check( );
       }
S25:  unsigned curCoins, alVend, totCoins ;
S26:   Dispenser d1 ;
S27:   Coinreturn c1;
E28:   public void check( ) {
S29:     if ( (curCoins < 2 ) && ( alVend) )
S30:          println ( "User can vend for free");
       }
    }
```

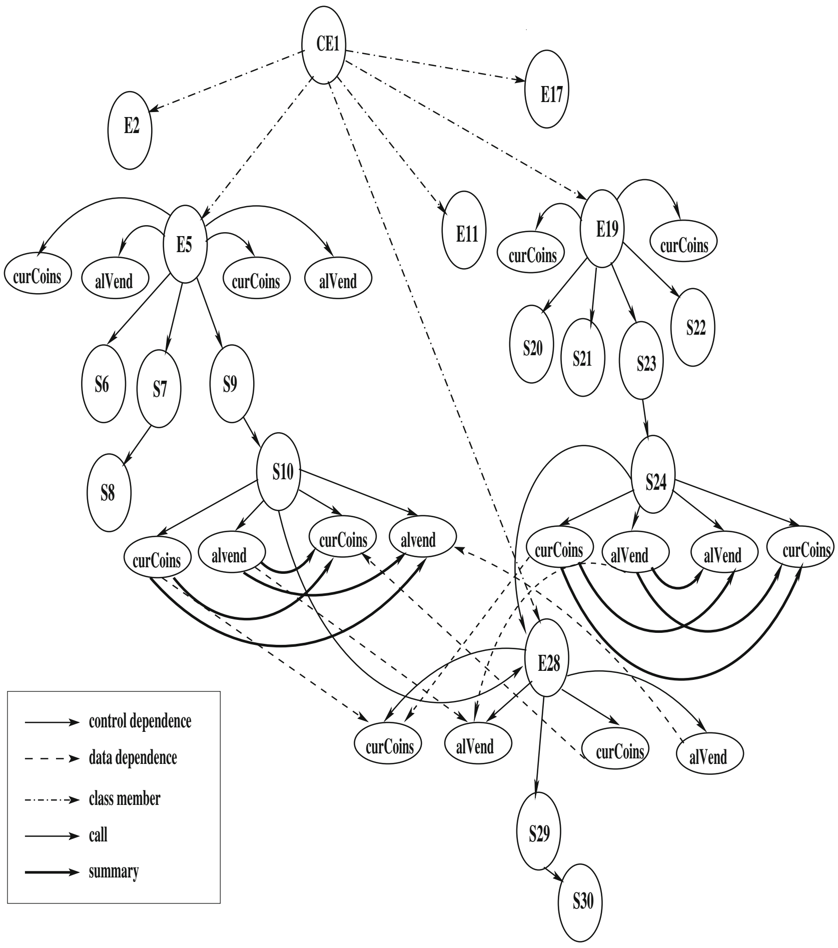

Fig. 2 A partial ESDG representation of the class *Coinbox* as given in Fig. 1

Table 1 Test suite and faults detected

Test cases	Faults							
	1	2	3	4	5	6	7	8
a	X							
b	X		X	X	X			
c	X	X	X	X	X			
d				X		X		
e						X	X	X

program contains eight faults and are detected by those test cases, as shown in the Table 1. Consider two ordering of these test cases, order O_1: *a–b–c–d–e*, and order O_2: *c–e–b–d–a*. Figure 3a, b show the percentage of faults detected versus the fraction of the test suite used, for these two orders O_1 and O_2 respectively. The area under the curve represents the average of the percentage of faults detected.

In case of test order O_1, after the execution of test case *a*, one of the eight faults is detected, thus only 12.5 % of faults have been detected after 0.2 % of test cases in test order O_1 has been used. But, after running test case *b*, four faults are detected and hence 50 % faults have been detected. In contrast, test order O_2 detects faults much earlier than test order O_1. In the test order O_2, first 0.2 % of test cases detect 62.5 % of the faults and 0.4 % of the test cases in the order detect 100 %. Test order O_1 has an APFD of 50 %. As can be observed from Fig. 3, test order O_2 detects faults much faster than O_1, and has an APFD of 70 %.

2.5 Coupling factor (COF) metric

The COF metric is an object-oriented design model metric from MOOD metric set [24]. The COF metric is used to measure the coupling between classes. The COF metric is calculated using the following formula:

$$COF = \frac{\sum_{i=1}^{TC} \lfloor \sum_{j=1}^{TC} is\_client(C_i, C_j) \rfloor}{TC^2 - TC} \tag{2}$$

where,

$$is\_client(C_i, C_j) = \begin{cases} 1 & \text{if } (C_i \Rightarrow C_j) \wedge (C_i \neq C_j) \\ 0 & \text{otherwise.} \end{cases}$$

C_i and C_j represent the client and supplier class respectively and TC represents total number of classes.

The client–supplier relation, represented by $C_i \Rightarrow C_j$, means that C_i contains at least one non-inheritance reference to a method or attribute of class C_j. The COF numerator represents the actual number of coupling not imputable to inheritance [25–28].

Coupling relations increase complexity of the system and reduce encapsulation as well as reusability of the system. However, for a given application, classes must cooperate to achieve some kind of functionality.

3 Related work

For ordering test cases of procedural programs several approaches have been proposed in the literature based on different criteria. Rothermel et al. developed a family of techniques for TCP based on several structural coverage criteria such as statement coverage, branch coverage, and the probability of exposing known faults [21, 29] etc. Rothermel and coworkers [8] used APFD metric to measure the effectiveness of their techniques. The APFD measure, however, gives better results under two assumptions: all faults are of equal severity, and all test cases have equal costs. Elbaum et al. [8] extended the empirical study reported by Rothermel et al. and additionally considered the coverage criterion at the function level.

The total and the additional coverage strategies are two widely-investigated greedy strategies for test-case prioritization. However, Rothermel et al.'s [21] empirical studies indicate that the greedy strategies may not always produce

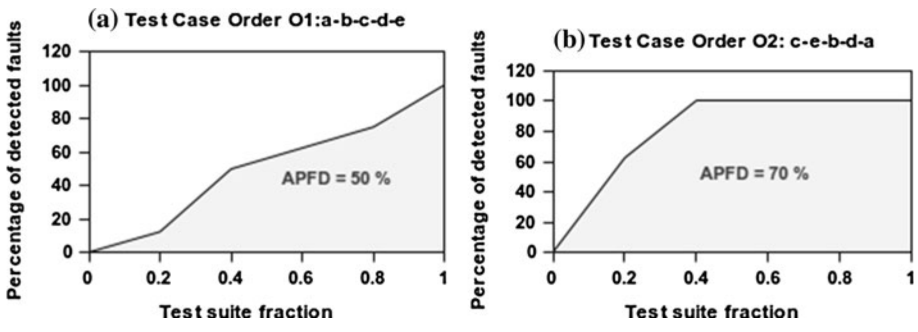

Fig. 3 Illustration of APFD measure

the optimal ordering of test cases. Li et al. [30] further proposed another greedy strategy that is, the two-optimal strategy (based on the k-optimal greedy algorithm [31]) and two meta-heuristic search strategies [32] to prioritize regression test cases.

Statement and function-level TCP techniques are considered in [8, 10, 21, 22] and these do not consider the modified condition/decision coverage (MC/DC) criterion [33]. The MC/DC is a stricter form of decision (or branch) coverage criterion which requires each condition in a decision is to be considered independently in order to reflect its effect on the outcome of the decision.

Elbaum et al. [10, 34] and Park et al. [35] studied the impacts of test costs and fault severities on TCP. Elbaum et al. proposed a cost-cognizant metric to evaluate the effectiveness of their approach called $APFD_c$ [10, 22]. This metric accounts for varying test case and fault costs during prioritization of regression test cases. In their work, the authors considered functional-level coverage [8] adapted to testing cost and severity level of faults. Park et al. [35] proposed a TCP technique based on estimation of test costs and fault severities that have been obtained using historical information.

Walcott et al. studied the problem of time-aware test-case prioritization and they considered an explicit time budget to carry out regression testing [11]. Walcott et al. used a genetic algorithm-based approach and the authors compared their approach with initial ordering, reverse ordering, random prioritization and fault-aware prioritization techniques.

Another approach for prioritizing regression test cases based on coverage of a relevant slice of the output of a test case was reported by Jeffrey and Gupta [23]. Their goal of prioritizing test cases was to achieve higher rates of fault detection. They defined a relevant slice as the set of statements that influence, or can influence the output of a program when run on a regression test case. Jeffrey and Gupta [23] observed that if a modification to the program affects the output of a regression test case, then it must affect some computation in the relevant slice for that test case. Based on this observation, they prioritized test cases by assigning higher weights to test cases with larger number of statements and branches in its relevant slice, in addition to the statements exercised by it. However, to the best of our knowledge, no work has been reported in the literature on regression TCP of object-oriented programs that considers coverage of various object-oriented features.

4 S-RTP: our proposed approach

We have named our proposed approach for regression test prioritization as slice based-regression test prioritization (S-RTP). Our technique first determines the affected nodes in ESDG model based on an analysis of control and data dependencies as well as dependencies arising from object-relations and then prioritizes regression test cases based on the number of affected nodes exercised by a test case.

The maintenance phase consists of multiple maintenance cycles as mentioned in Sect. 2.2 There can be many regression testing cycles in each maintenance cycle and in each regression testing cycle, TCP is an important activity.

Figure 4 shows the important steps of our approach carried out in first regression test prioritization cycle using an activity diagram. As shown in Fig. 4, the important activities include constructing the ESDG model, collecting test coverage information and marking the test coverage information on ESDG model and are not repeated for subsequent test prioritization cycles in our approach. We now describe the different activities that are carried out during first regression test cycle in S-RTP in more detail.

– *Construct ESDG model*: In this step, the ESDG model as described in Sect. 2.1 for the original program P is constructed using a technique similar to that reported in [14].
– *Identify changes*: The changes between P and the modified program P' are identified through semantic analysis and the identified statement-level changes are kept in a file named as *differ*. Each entry in the *differ* file contains the changed statements and the line number in both P and P'. This is shown by the data store *Change information* in Fig. 4.
– *Instrument and execute the program*: In this step, P is instrumented by inserting print statements at the basic block level. These print statements serve to collect test coverage information. The instrumented code is executed with the original test suite T to generate information pertaining to the specific statements that are executed for each test case that is the execution trace for each test case. The instrumented statements are stripped during regression test execution, so that the original program behavior is not affected. Generating the test coverage information is an one-time activity for a given program during one testing cycle, and need not be repeated during the subsequent regression testing sessions in that cycle. The test coverage information generated in this step is logged in a file denoted by *CovgInfo*, and is saved for later processing. This is represented by the data store *Test coverage information* in Fig. 4.
– *Mark ESDG model*: In this step, the test coverage information is marked on ESDG model. Marking an ESDG model involves tagging to each node in the ESDG model information about the test cases that execute the corresponding program statement.

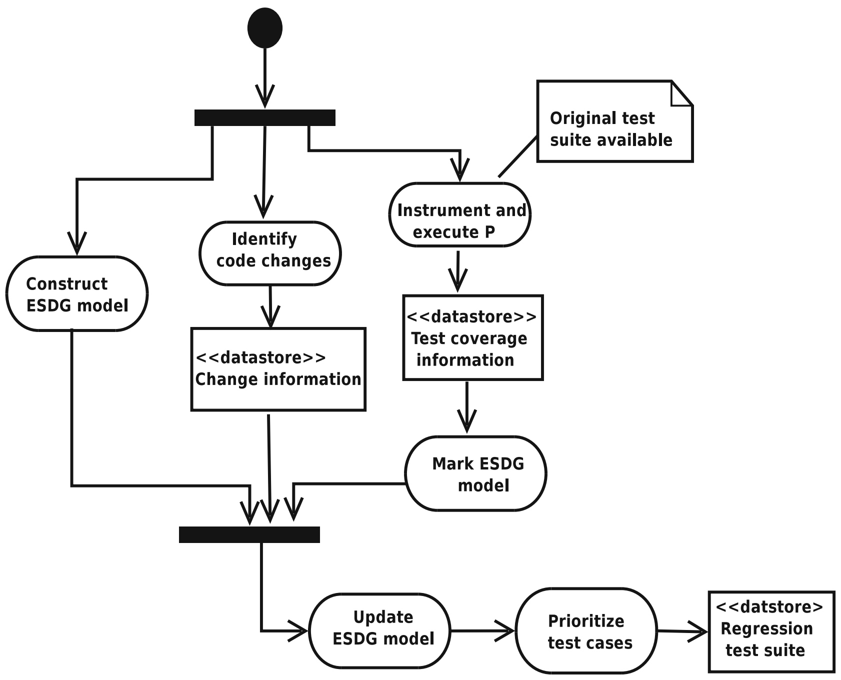
Original test suite available
Instrument and execute P
Identify code changes
Construct ESDG model
<<datastore>> Change information
<<datastore>> Test coverage information
Mark ESDG model
Update ESDG model
Prioritize test cases
<<datstore> Regression test suite

Fig. 4 Activity diagram representation of S-RTP

– *Update ESDG model*: In this step, the model constructed for P is updated using information from *differ* file during each regression testing cycle to make it correspond to the modified program P' and the updated marked ESDG model is denoted by M_u. A description of the methodology for updation of the ESDG model with change information is discussed in more detail in Sect. 4.2

– *Prioritize test cases*: In this step, regression test cases are prioritized based on the number of affected nodes exercised by a test case. The prioritized set of regression test cases are represented by the data store *Regression test suite* in Fig. 4. This step is discussed in more detail in Sect. 4.3

4.1 Types of program changes

Without any loss of generality the changes made to a program can be assumed to be one or more of the following types: (1) *addition* (2) *deletion* or (3) *modification* of a statement. A modification operation can be expressed as a deletion operation followed by an addition operation. Hence, we assume that addition and deletion are the two basic types of changes that can be made to a program.

Any change made to a program may affect the dependency relations already existing in the program. Addition of a statement to a program requires creation of a new node and one or more edges in the ESDG model and may also require deletion of some existing edges. This is explained in the following with an example. Consider the code segment shown in Fig. 5a and the corresponding partial ESDG model consisting of control (solid) and data dependence (dotted) edges. In Fig. 5a, S_5 and S_6 are two program statements in the method m. In the partial ESDG, these two nodes are also denoted by S_5 and S_6 and are connected to the method entry node S_0 by *control* dependence edges. The partial ESDG model shows that the nodes S_5 and S_6 are data dependent on S_1. Now, suppose we add a statement S_j to the original code segment as has been shown in Fig. 5b. After addition of statement S_j, the statement S_6 is no longer data dependent on S_1 and instead is data dependent on statement S_j. Due to addition of statement S_j, the data dependence edge between nodes S_1 and S_6 needs to be deleted, and two new data dependence edges need to be added between the pairs of nodes S_1 and S_5 and S_j and S_6. Also, a control dependence edge from *method entry* node S_0 to the node S_j is introduced.

Addition or deletion of one or more statements can affect the dependencies existing among program statements, besides those in which the added or deleted statement is directly participate. For example, if a statement defining a variable is deleted, then it would affect the dependency structure of the dependent statements and these statements may have dependency on some other statement(s). So, in case of deletion of a statement, before deleting a statement it is required to identify and mark all those program statements that are data and control dependent or dependent due to object-relations on the deleted statement. Before a node(s) corresponding to the deleted statement is deleted, the other nodes in ESDG model that

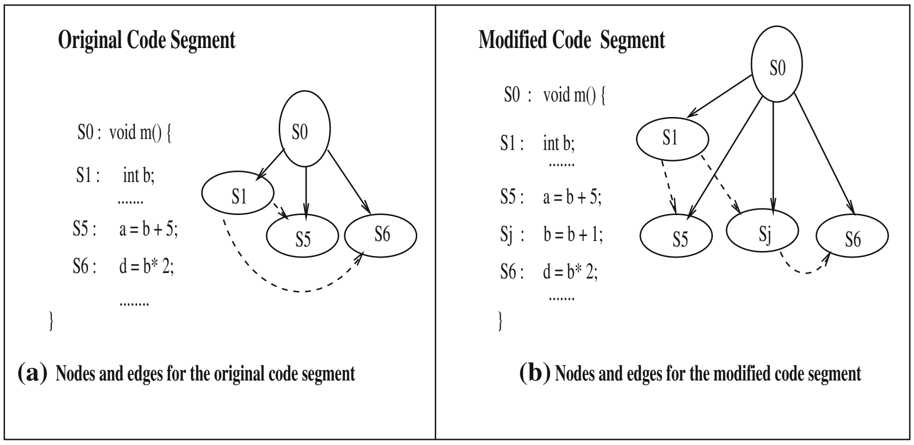

Fig. 5 Effect of addition of a statement on data and control dependencies

are data or control dependent or dependent due to object-relations on the deleted node(s) are identified and are marked as *affected*. Then the node corresponding to the deleted statement is deleted. Subsequently, the *in* and *out* edges from the deleted node are also deleted and new dependency edges as required are created.

We give an example of the effect of the deletion of a statement in Fig. 6. Figure 6a shows a code segment and the corresponding partial ESDG model consisting of control (solid) and data dependence (dotted) edges. In the partial ESDG model, the node S_q is data dependent on node S_p and the node S_r is data dependent on nodes S_p and S_q in the original code. Suppose the statement S_q is deleted as shown in Fig. 6b. Before deletion of the node S_q, the node S_r is marked as *affected*, as the node S_r is data dependent on node S_q. Now, due to deletion of the statement S_q, an additional data dependence edge is also introduced between the nodes S_1 and S_r. Finally, the node S_q and all *in* and *out* edges of S_q are deleted from the partial ESDG model.

4.2 Updation of an ESDG model

This section describes our model updation technique by which the constructed ESDG model is updated when a change is made to the corresponding program. We use the term size of the ESDG to mean an upper bound on the number of vertices in an ESDG. Construction of an ESDG model each time a change is made to a program can incur significant overhead especially for large programs. To overcome this problem, instead of creating an ESDG model each time changes are made, we update the original ESDG model to reflect the changes made to it.

For every addition of a statement, usually a single node is created. But for statements such as method call (also called *call-site*) or called method, more than one node needs to be created in an ESDG model. If it is a method call statement, then in addition to a node for calling method, we create one node for each *actual-in* and *actual-out* parameter. The node for calling method is connected to the *method entry* node by a *control dependence* edge. Then the *call-site* node is connected to its *method entry* node by a *call* edge. The *actual-in* parameter nodes are connected to the *formal-in* parameters nodes in the called method by *parameter-in* edges. The *formal-out* node of the called method is connected to the *actual-out* node by a *parameter-out* edge. The return statement node of a method is connected to its corresponding method call node using a *parameter-out* edge. If the value associated with a *actual-in*

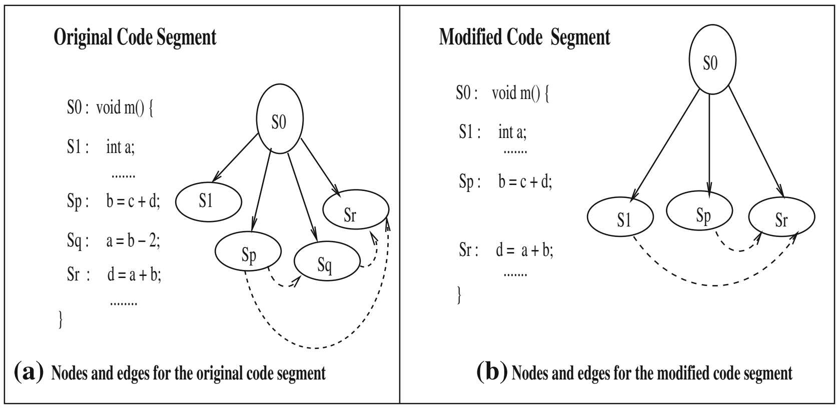

Fig. 6 Effect of deletion of a statement on data and control dependencies

```
S0 :  public class ABC {
S1 :           int a, b ;
S2 :           public ABC ( ) {
S3 :                    a = 4;
S4 :                    b = 18; }
S5 :           public int incre ( y ) {
S6 :                    add (y, 1) ; } // added statement
S7 :           public int add ( int a, int b) {
S8 :                    a = a + b;  }
       }
```

Fig. 7 Example of method call

node affects the value associated with a *actual-out* node, then a *summary* edge is added to connect *actual-in* node to *actual-out* node. The *summary* edge represents the transitive dependency across a call-site caused by either control or data dependencies or both and is used for inter-procedural slicing.

Next, we give an example of the effect of addition of a *method call* statement in the ESDG model. Consider the code segment as shown in Fig. 7. In the code segment, S_6 is a *method call* statement. The corresponding partial ESDG model as shown in Fig. 8 represents the addition of different types of nodes and edges due to addition of a method call statement. In the partial ESDG model, square represent the *method entry* nodes, circle represent statement nodes and ellipses represent the parameter nodes. In Fig. 8, the *method call* node, *method entry* node and *simple statement* nodes are same as statement number in the code segment. The parameter nodes are named according to the parameters passed to the methods as given in the code segment. When the method call statement S_6 is added in the code segment as shown in Fig. 7, the node S_6 and the parameter nodes $a\_in = y$, $b\_in = 1$ and $y = a\_out$ are added in the ESDG model. These are connected to S_6 by *control dependence* edges. Then S_6 is connected to the corresponding *method entry* node S_7 by a *call* edge. The *actual-in* parameters that is $a\_in = y$ and $b\_in = 1$ are connected to the corresponding *formal-in* parameter nodes $a = a\_in$ and $b = b\_in$ in S_7 by *parameter-in* edges. The formal-out parameter node $a\_out = a$ of the called method S_7 is connected to the *actual-out* node $y = a\_out$ by a *parameter-out* edge. As the *actual-in* parameter nodes in S_6 affects the value of *actual-out* parameter node $y = a\_out$, so *summary edges* are added from $a\_in = y$ to $y = a\_out$ and $b\_in = 1$ to $y = a\_out$.

For a *method entry* statement, a *method entry* node is created and is connected to its *class entry* node by a *class member* edge. Similarly, for a simple statement such as assignment, conditional etc., a *simple statement* node is created, then that *simple statement* node is connected with its *method entry* node by a *control dependence* edge.

In case of deletion of a statement, we delete the corresponding node(or nodes) from ESDG model. But, before deleting the node(or nodes), we determine the nodes that are dependent on the deleted node on account of either control and data dependencies or object-relations such as association or inheritance by slicing, where each deleted node is used as slicing criteria and are marked as *affected*. After marking these dependent nodes as *affected*, we delete all *in* and *out* edges from the deleted node and then the node corresponding to the deleted statement is deleted from ESDG model.

The changes between the programs P and P' are also marked on the ESDG model by using the information from the *differ* file. This is done as follows: The statements which are deleted from P, the nodes in M which were directly data or control dependent on the deleted nodes are affected due to deletion, and are tagged as deleted. These nodes are determined by constructing slices on ESDG model where each deleted node is used as a slicing criterion. Once the model M has been updated to correspond to P', for each statement added or modified in P', the nodes corresponding to the changed statements in the updated model M_u are tagged as changed. We refer to both the set of tagged nodes as *Tagged*.

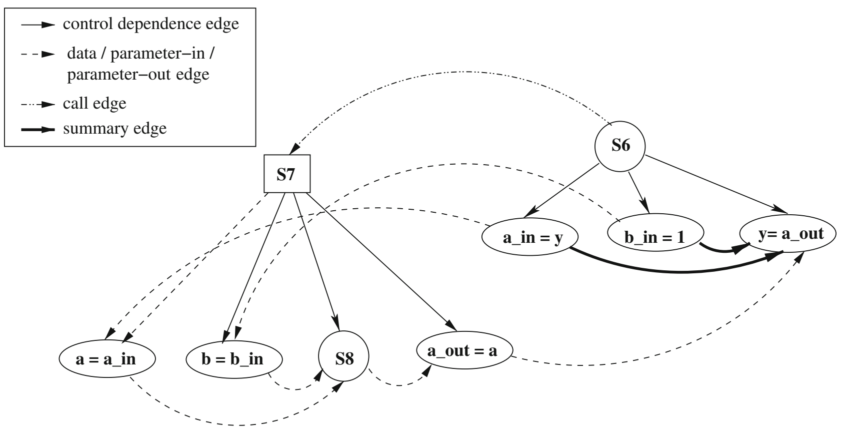

Fig. 8 Effect of addition of a method call statement as shown in Fig. 7

4.3 TCP using ESDG model

For prioritization of test cases, first forward slices are constructed on ESDG model by using each added, modified, and *affected* node as the slicing criterion. Construction of the forward slice helps to determine the nodes that have been affected due to specific modifications made to the code. First all those test cases that exercise the affected model elements are found from forward slice. These are in the regression test suite. Then the test cases are assigned priority based on the number of affected nodes exercised by each selected test case. The pseudocode to determine the affected nodes in ESDG model is given in Algorithm 1. We have named this algorithm *ESDGSlice*. The working of the algorithm *ESDGSlice* is described in the following.

Algorithm 1: Pseudocode to determine the affected nodes by computing ESDG model slice.

```
Input: $M_u$, $Tagged$
Output: $AffectedSet$
1 procedure $ESDGSlice(M_u, Tagged, AffectedSet)$
2 $AffectedSet \leftarrow NULL$
3 for each node $n$ in $Tagged$ do
4     Find the nodes that are data or control dependent or
      dependent due to object-relations such as inheritance and
      association on $n \in M_u$
5     $AffectedSet = AffectedSet \bigcup$ {all nodes that are data or
      control dependent or dependent due to object-relations
      such as inheritance or association on $n$}
6 end
```

To determine the affected nodes denoted by *AffectedSet*, we first compute the forward slice on updated marked ESDG model. Our forward slicing algorithm is based on the two-pass graph reachability algorithm proposed by Horwitz et al. [13]. *ESDGSlice* takes M_u and the set of *tagged* nodes as input, and produces the set of affected nodes denoted by *AffectedSet* as output. *ESDGSlice* computes the set of all affected nodes and the steps are given in lines 3–5 in Algorithm 1. Slicing the ESDG model helps to identify the set of model elements that may get affected due to the modifications.

After all the affected nodes in ESDG model are identified through forward slicing, a test case that exercises a higher number of affected nodes is given higher priority. The reason behind it is that, a test case which covers higher number of affected nodes in the dependency model has a more chance to detect an error(s) than a test case which covers lesser number of affected nodes.

Let T' be the set of test cases that executes one or more affected nodes in the ESDG model. Initially T' is null and let WT be weight for the test cases in T'. $WT = \{wt_1, wt_2, \ldots, wt_n\}$ and $wt_i = 0$ for all i initially and n represents the nth test case in T'. The pseudocode to prioritize the regression test cases is given in Algorithm 2. The algorithm to prioritize test cases is named as *TestPrior* and the input to *TestPrior* is the M_u, *WT* and the *AffectedSet*. The output of *TestPrior* is the prioritized set of regression test cases denoted by T'. To prioritize test cases, for each node n_i in *AffectedSet*, add the test case t_j executing n_i to T' and the weight of t_j denoted by wt_j is increased by 1. Step 4 in Algorithm 2 is trivial since the information about which test cases execute which model elements is already marked on the corresponding node in the ESDG model. Finally, the test cases in T' are sorted according to their decreasing weights in the list *WT*.

Algorithm 2: Pseudocode to prioritize regression test cases.

```
Input: $M_u$, $AffectedSet$
Output: $T'$
1  procedure TestPrior($M_u$, $WT$, $AffectedSet$, $T'$)
2  $T' \leftarrow NULL$
3  $WT \leftarrow NULL$
4  for each node $n_i \in AffectedSet$ do
5      for each test case $t_j$ that executes the node $n_i$ in $M_u$ do
6          Add $t_j$ to $T'$
7          $wt_j = wt_j + 1$
8      end
9  end
10 Sort the test cases in $T'$ in the decreasing order of their weights
   in the list $WT$
```

5 Experimental study

In this Section, we first discuss a prototype tool Regression-TEST Prioritization (R-TESTPrio) that we have developed to implement our approach. Next, we explain the experimental setups made and the results achieved in the subsequent subsections.

In order to measure the effectiveness of our approach, we have used the APFD metric [21]. In our experimental study, we first prioritize the test cases using S-RTP. Then APFD values are calculated for the prioritized test suite of each of the considered program. We have compared our approach with Smith et al.'s approach [36]. We have implemented their approach only for non-overlap-aware greedy algorithm [37] to prioritize test cases. For their approach, we build a calling context tree (CCT) for a program as described in [38]. The call tree constructor uses instrumentation probes to create the CCT. After the call tree is initialized, the tree constructor executes a probe before and after the execution of each method and test case.

We prioritize the test cases using Smith et al.'s approach based on coverage of call tree paths by a test case and calculate the APFD metric for Smith et al.'s approach [36] for the prioritized test suite. In order to evaluate the effectiveness of our approach, we compare the APFD values for each of the considered program as calculated using S-RTP and Smith et al.'s approach.

5.1 R-TESTPrio: a prototype implementation of S-RTP

R-TESTPrio has been developed using the programming language Java on a Microsoft Windows XP environment. The code size of R-TESTPrio is approximately 14,000 LOC, excluding the external packages that are used in implementation of S-RTP technique. The user interface of R-TESTPrio is developed using Java Swing.

We now briefly mention the various open source software packages used to implement our approach and the different components of R-TESTPrio in the following.

Open source software packages used: The tool R-TESTPrio is developed using the following open source software packages: Eclipse [39], ANTLR [40] and Graphviz [41]. We have used eclipse as an IDE and ANTLR as the parser generator, and have adapted ANTLR grammar file for Java language [42]. ANTLR works as an Eclipse plug-in. We have used Graphviz to graphically present the ESDG model constructed by R-TESTPrio. We have used an Eclipse plug-in version of an open source software μJava (MuJava), a mutation system for Java programs to generate mutants [43]. The eclipse plug in version of MuJava is known as MuClipse. (An open source mutation testing plug-in for Eclipse http://muclipse.sourceforge.net/). It provides a bridge between the MuJava API and Eclipse. MuClipse automatically generates mutants for both traditional and class-level mutation operators.

Components of R-TESTPrio: Figure 9 shows a schematic representation of our prototype tool. R-TEST-Prio comprises of 5 major components, namely, *ESDG Model Constructor, Test Coverage Generator, ESDG Model Marker, ESDG Model Updater* and *Test Case Prioritizer*. In Fig. 9, rectangles represent the different components of R-TESTPrio and the ellipses represent data or artifacts. Initially, R-TESTPrio takes P, P', *Change Information* and the initial test suite T as input. In R-TESTPrio, the input to a component is used as an output of an another component. The component *Test Coverage Generator* produces output *Test Coverage Information* that is used as an input to the component *ESDG Model Marker* in R-TESTPrio. The *Change Information* (described in Sect. 4) is used as an input to the component *ESDG Model Updater* to update the original model M.

We describe the working of different components of R-TESTPrio in the following.

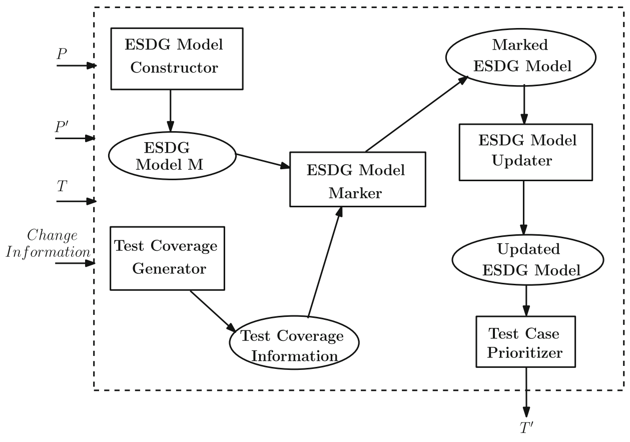

Fig. 9 Schematic of design of R-TESTPrio

– *ESDG Model Constructor: ESDG Model Constructor* is developed in Java using Eclipse IDE [39] and ANTLR tool [40]. *ESDG model constructor* takes as input a Java program and constructs the ESDG model for the program.
– *Test Coverage Generator:* This component generates test coverage information by executing the input program with all test cases from T. The generated test coverage information is stored in file *CovgInfo* that lists the line numbers executed by each test case in T.
– *ESDG Model Marker:* This component of R-TESTPrio stores the test coverage information from *CovgInfo* in the ESDG model. This approach of storing test coverage information in the model itself circumvents the use of file storage and improves the efficiency of our approach. To store the test coverage information, ESDG model is traversed to find out the corresponding node(s) modeling each program statement s in P. Then, for each node corresponding to a program statement s, *ESDG Model Marker* stores the list of test cases that execute the node corresponding to s.
– *ESDG Model Updater: ESDG Model Updater* component is used to update the ESDG model. This component takes the *differ* file as external input. The updated model reflects the changes that are made to P.
– *Test Case Prioritizer:* This component prioritizes the test cases based on the number of affected nodes exercised by different test cases. The output of this component is a file that contains the identifiers of regression test cases in priority order.

5.2 Experimental setup

We have used seven Java programs in our experimentation: elevator control (EC), cruise control (CrC), binary search tree (BST), automated teller machine (ATM), power window controller (PWC), ordered set (OrS) and vending machine (VM). The characteristics of these programs have been given in Table 2.

The sizes of the considered programs ranged from 229 to 943 lines of source code and the sizes of the test suites ranged from 17 to 42 test cases. The first column in Table 2 shows the programs that were used in the experimental studies. Columns 2 and 3 show the sizes of the programs and the number of classes for our example programs respectively. The total number of test cases in each of the considered program is given in column 4 and the total number of mutation faults of each program in column 5. The test cases for these considered programs were designed by us using black-box test case design techniques.

The programs that we have considered in our study were of moderate size and faults were generated through code mutation using MuClipse (An open source mutation testing plug-in for Eclipse http://muclipse.sourceforge.net/). MuClipse provides an API to generate mutants. It allows one to select the types of mutation operators to be employed. MuClipse uses two types of mutation operators, class level and method level. Class level operators are specialized to object-oriented faults, whereas the method level operators are the same as traditional mutation operators. The specific class level and the traditional operators that are supported by MuClipse are discussed in [44–47] respectively.

We have used COF metric to determine the interdependencies between classes and COF metric is described in more detail in Sect. 2.5 The programs having higher COF value indicates higher interdependency. We have categorized our considered programs into 3 types, that is, low, moderate and high based on this aspect and is shown in Table 3. The programs having COF value up to 1 are considered as low, more than 1 and up to 2 as moderate and more than 2 as high in our approach. In Table 3, column 1 represents the program names, columns 2 and 3 represent the COF value for each program and the category of programs respectively.

5.3 Results

In this section, we present the results obtained from our experimental studies. Table 4 summarizes the main results.

Table 3 Categories of programs

Program name	COF	Program categories
BST	0.58	
ATM	0.83	Low
OrS	0.5	
EC	1.06	Moderate
VM	1.8	
CrC	2.2	High
PWC	2.5	

The columns 2 and 3 of Table 4 show the APFD for R-TESTPrio and Smith et al.'s approach in percentage for each of our considered programs respectively and column 4 represents the percentage increase in APFD for R-TESTPrio over that of Smith et al.'s approach.

From Table 4 it can be observed that for all the considered programs, APFD of R-TESTPrio is on the average 25.70 % higher as compared to Smith et al.'s approach. This increase may be due to the fact that, our approach considers several intrinsic features of an object-oriented program such as class, dynamic binding and object-relations like inheritance, association and polymorphism for prioritization.

The results of Table 4 have been presented in the form of a bar graph in Fig. 10. In Fig. 10, the *x*-axis represents the programs used in our experiments and *y*-axis represents the APFD values for all programs using S-RTP and Smith et al.'s approach.

For the considered programs, we have also analyzed the behavior of the program with increase in the number of mutation faults. For analysis of individual program, we choose one program from each considered category as given in Table 3 in our experimental studies. We have considered the programs CrC, EC and BST with high, moderate and low COF values respectively. We calculate the APFD values of each of the program with increase in the number of mutation faults and analyze the trend in APFD values for different categories of programs.

For each of the considered programs CrC, EC and BST, we determine the APFD values using R-TESTPrio. We

Table 2 Summary of programs used in experimentation

Program name	Size (LOCs)	Classes	Test cases	Mutation faults
EC	943	8	18	72
CrC	649	4	42	46
BST	564	4	17	31
ATM	623	3	25	38
PWC	720	6	36	81
OrS	229	2	18	42
VM	471	3	25	53

Table 4 Summary of experimental results

Program name	APFD for R-TESTPrio	APFD for Smith et al.'s [36] approach	% Increase
EC	86.2	68.6	25.65
CrC	79.6	60.4	28.01
BST	83.4	64.1	30.10
ATM	81.1	63.7	27.31
PWC	82.2	66.7	23.23
OrS	85.2	68.7	24.01
VM	80.4	66.1	21.63

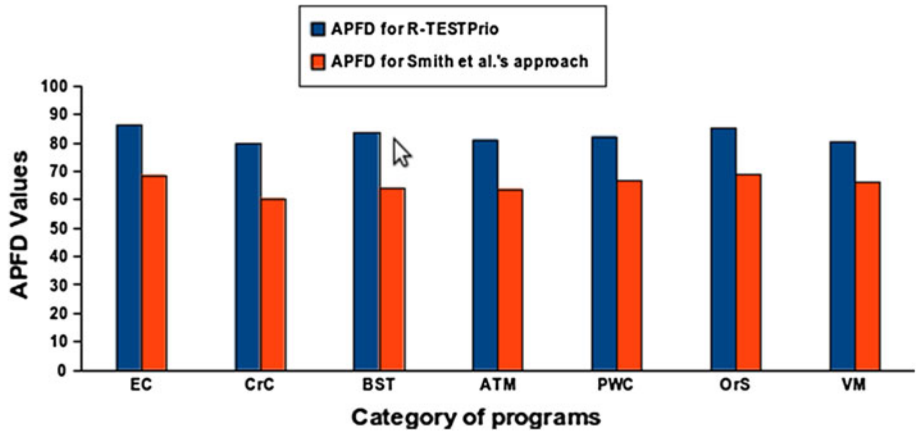

Fig. 10 Comparison of APFD values for S-RTP and Smith et al.'s approach

increase the number of mutation faults from 5 % to 25 % for all three considered programs. The Tables 5, 6 and 7 show the results obtained from our experimental study for the analysis of the specific programs CrC, EC and BST. In the Tables 5, 6 and 7, the first column represents the program, column 2 represents the % increase in the number of mutation faults, and column 3 shows the APFD values in percentage while using R-TESTPrio and increasing the number of mutation faults.

The results Tables 5, 6 and 7 have been presented in the form of line graphs in Figs. 11, 12 and 13 respectively. In

Table 5 APFD values for the program cruise controller using R-TESTPrio

Program name	% of mutation faults	APFD for R-TESTPrio
CrC	5	38.3
	10	39.5
	15	42.7
	20	49.1
	25	60.3

Table 6 APFD values for the program elevator controller using R-TESTPrio

Program name	% of mutation faults	APFD for R-TESTPrio
EC	5	15.2
	10	16.1
	15	17.4
	20	20.4
	25	23.9

Table 7 APFD values for the program binary search tree using R-TESTPrio

Program name	% of mutation faults	APFD for R-TESTPrio
BST	5	21.5
	10	22.7
	15	24.3
	20	26.5
	25	30.2

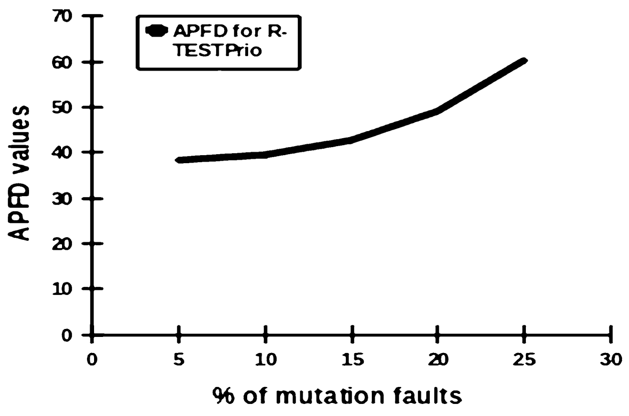

Fig. 11 APFD values for the program CrC

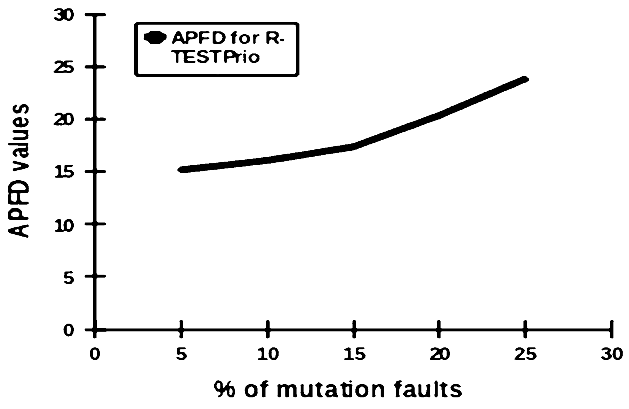

Fig. 12 APFD values for the program EC

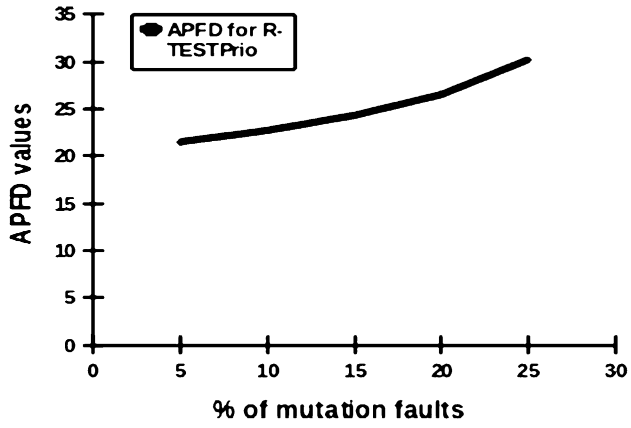

Fig. 13 APFD values for the program BST

these graphs, x-axis represents the % of mutation faults and y-axis represents the APFD values for the programs using R-TESTPrio.

From Tables 5, 6 and 7 it can be observed that, for all the three categories of programs, with increase in the number of mutation faults, APFD values using R-TESTPrio increases. This increase may be due to the fact that, with increase in the number of faults, the prioritization order using R-TESTPrio increases the rate of fault detection. But, the percentage increase in the APFD values with increase in the number of mutation faults is higher for CrC as compared to EC and BST and EC is higher as compared to BST. This may be due to the fact that, with increase in the interdependency between classes, the percentage increase in the APFD values for the programs with high interdependency is higher as compared to the programs with low interdependency.

6 Comparison with related work

Literature study indicate that, most of the proposed regression TCP techniques order test cases based on greedy strategies using coverage criteria such as statement coverage, additional statement coverage, and function coverage etc. for procedural programs. But, these techniques are not suitable for object-oriented programs as statement and function coverage are not very meaningful due to object-oriented features such as inheritance and dynamic binding [12]. According to Thuy, exercising a single binding of a polymorphic call by a test case is not sufficient and it is required to exercise all possible bindings [48]. Further, a regression test prioritization technique for object-oriented programs needs to consider dependencies introduced by various object-relations.

A regression test reduction and prioritization approach using call trees was proposed in the literature for object-oriented programs by Smith et al. [36]. To the best of our knowledge no directly comparable work on regression TCP of object-oriented programs based on coverage of object-oriented features has been reported in the literature. Consequently, we have compared our approach with Smith et al.'s approach. The results of our experimental study indicate that, the APFD metric values for R-TESTPrio is on an average higher by about 25.70 % over Smith et al.'s approach.

7 Conclusion

We have proposed an approach for regression TCP that orders test cases based on the number of affected model elements exercised by a test case in the program model of object-oriented programs. We determine the affected parts of the program based on consideration of dependencies arising on account of object-relations in addition to the data and control dependencies. For this, we construct an ESDG model and the affected elements are determined based on slicing the ESDG model. Our approach can be generalized to consider other object-oriented programming features such as exceptions, threads, interfaces and class libraries by suitably augmenting the graph model. We have evaluated the effectiveness of our proposed regression TCP technique using several moderate sized Java programs. The results of our study show that our approach on an average achieves an increase in the APFD value by 25.70 % compared to a related approach.

References

1. Harrold M, Jones J, Li T, Liang D, Orso A (2001) Regression test selection for java software. In: Proceedings of the 16th ACM SIGPLAN conference on object-oriented programming, systems, languages and applications, pp 312–326
2. Rothermel G, Harrold M, Dedhia J (2000) Regression test selection for C++ software. J Softw Test Verif Reliab 10(6):77–109
3. Rothermel G, Harrold MJ (1994) Selecting regression tests for object-oriented software. In: Proceedings of the International Conference on Software Maintenance, Victoria, pp 14–25
4. Harrold M, Gupta R, Soffa M (1993) A methodology for controlling the size of a test suite. ACM Trans Softw Eng Methodol 2(3):270–285
5. Korel B, Tahat L, Vaysburg B (2002) Model based regression test reduction using dependence analysis. In: Proceedings of IEEE International Conference on Software Maintenance, pp 214–223
6. Ma X, Sheng B, Ye C (2005) Advanced parallel processing technologies, chapter test-suite reduction using genetic algorithm. Springer, Berlin
7. Wong WE, Horgan JR, London S, Mathur AP (1998) Effect of test set minimization on fault detection effectiveness. Softw Pract Experience 28(4):347–369
8. Elbaum S, Malishevsky A, Rothermel G (2002) Test case prioritization: a family of empirical studies. IEEE Trans Softw Eng 28(2):159–182

9. Elbaum S, Rothermel G, Kanduri S, Malishevsky A (2004) Selecting a cost-effective test case prioritization technique. Softw Qual Control 12(3):185–210
10. Malishevsky A, Ruthruff J, Rothermel G, Elbaum S (2006) Cost-cognizant test case prioritization. Technical Report TR-UNL-CSE-2006-0004
11. Walcott K, Soffa M, Kapfhammer G, Roos R (2006) Time-aware test suite prioritization. In: Proceedings of the 2006 International Symposium on Software Testing and Analysis, pp 1–12
12. Binder RV (2003) Testing object-oriented systems: models, patterns, and tools. Addison-Wesley, Menlo Park
13. Horwitz S, Reps T, Binkley D (1990) Interprocedural slicing using dependence graphs. Trans Program Lang Syst 12(1):26–60
14. Larsen L, Harrold M (1996) Slicing object-oriented software. In: Proceedings of 18th IEEE International Conference on Software Engineering
15. Ferrante J, Ottenstein KJ, Warren JD (1987) The program dependence graph and its use in optimization. ACM Trans Program Lang Syst 9(3):319–349
16. Website. http://www.bugzilla.org/. Accessed 30 July 2012
17. Weiser M (1981) Program slicing. In: ICSE'81: Proceedings of the 5th International Conference on Software Engineering, pp 439–449
18. Tip F (1995) A survey of program slicing techniques. J Program Lang 3:121–189
19. Gallagher K, Lyle J (1991) Using program slicing in software maintenance. IEEE Trans Softw Eng 17(8):751–761
20. Mund G, Mall R (2006) An efficient interprocedural dynamic slicing method. J Syst Softw 79(6):791–806
21. Rothermel G, Untch R, Chu C, Harrold M (2001) Prioritizing test cases for regression testing. IEEE Trans Softw Eng 27(10): 929–948
22. Elbaum S, Malishevsky A, Rothermel G (2001) Incorporating varying test costs and fault severities into test case prioritization. In: Proceedings of the 23rd International Conference on Software Engineering, pp 329–338, Ontario
23. Jeffrey D, Gupta N (2008) Experiments with test case prioritization using relevant slices. J Syst Softw 81:196–221
24. Brito e Abreu F (1995) The MOOD metrics set. In: Proceedings of ECOOP'95 Workshop on Metrics
25. Abreu F, Carapuca R (1994) Object-oriented software engineering: measuring and controlling the development process. In: Proceedings of the 4th International Conference on Software Quality, McLean, pp 231–247
26. Abreu F, Goulao M, Esteves R (1995) Toward the design quality evaluation of object-oriented software systems. In: Proceedings of the 5th International Conference on Software Quality, Austin
27. Abreu F, Melo W (1996) Evaluating the impact of object-oriented design on software quality. In: Proceedings of the 3rd International Software Metrics Symposium (METRICS'96). IEEE, Berlin, pp 90–99
28. Harrison R, Counsell S, Nithi R (1998) An evaluation of the MOOD set of object-oriented software metrics. JIEEE Trans Softw Eng 24(6):491–496
29. Rothermel G, Untch RH, Chu C, Harrold MJ (1999) Test case prioritization: an empirical study. In: Proceedings of ICSM, pp 179–188
30. Li Z, Harman M, Hierons R (2007) Search algorithms for regression test case prioritization. IEEE Trans Softw Eng 33(4):225–237
31. Lin S (1965) Computer solutions of the travelling salesman problem. Bell Syst Tech J 44(5):2245–2269
32. Reeves CR (1993) Modern heuristic techniques for combinatorial problems. Wiley. New York
33. Jones JA, Harrold MJ (2003) Test-suite reduction and prioritization for modified condition/decision coverage. IEEE Trans Softw Eng 29(3):195–209
34. Elbaum S, Malishevsky A, Rothermel G (2000) Prioritizing test cases for regression testing. In: Proceedings of ISSTA, ACM Press, pp 102–112
35. Park H, Ryu H, Baik J (2008) Historical value-based approach for cost-cognizant test case prioritization to improve the effectiveness of regression testing. In: Proceedings of SSIRI, pp 39–46
36. Smith A, Geiger J, Kapfhammer GM, Soffa ML (2007) Test suite reduction and prioritization with call trees. In: Proceedings of ASE'07, Atlanta, 4–9 Nov 2007
37. Rummel M, Kapfhammer GM, Thall A (2005) Towards the prioritization of regression test suite with data flow information. In: Proceedings of 20th SAC, pp 1499–1504
38. McMaster S, Memon A (2005) Call stack coverage for test suite reduction. In: Proceedings of 21st ICSM, pp 539–548
39. Eclipse. http://www.eclipse.org/. Accessed 16 May 2012
40. Antlr Parser Generator. http://www.antlr.org/. Accessed 14 May 2012
41. Graphviz (2012). http://www.graphviz.org/. Accessed 21 Aug 2012
42. Scott Wisniewski. ANTLR Java Grammar. http://www.antlr.org/grammar/list. Accessed 14 May 2012
43. Ma Y-S, Offutt J, Kwon Y-R (2005) MuJava: an automated class mutation system. J Softw Test Verif Reliab 15(2):97–133
44. Kim S, Clark J, McDermid J (2000) Class mutation: mutation testing for object-oriented programs. OOSS: Object-Oriented Software Systems
45. Ma Y-S, Kwon Y-R, Offutt J (2002) Inter-class mutation operators for Java. In: Proceedings of the 13th International Symposium on Software Reliability Engineering. IEEE Computer Society Press, Annapolis, pp 352–363
46. Agrawal H, DeMillo RA, Hathaway R, Hsu W, Wynne H, Krauser EW, Martin RJ, Spafford EH (1989) Design of mutant operators for the C programming language. In: Technical Report: SERC-TR-41-P, Software Engineering Research Center, Purdue University
47. Delamaro ME, Maldonado JC, Mathur AP (2001) Interface mutation: an approach to integration testing. IEEE Trans Softw Eng 27(3):228–247
48. Thuy NN (1992) Testability and unit tests in large object oriented software. In: Proceedings, 5th International Software Quality Week. Software Research, San Francisco