Ray M, Kumawat KL, Mohapatra DP. Source code prioritization using forward slicing for exposing critical elements in a program. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 26(2): 314–327 Mar. 2011. DOI 10.1007/s11390-011-9438-1

Source Code Prioritization Using Forward Slicing for Exposing Critical Elements in a Program

Mitrabinda Ray, Kanhaiya lal Kumawat, and Durga Prasad Mohapatra, *Member*, *IEEE*

Department of Computer Science and Engineering, National Institute of Technology, Rourkala, Orissa 769008, India

E-mail: mitrabindaray@yahoo.co.in; kk.nitrkl@gmail.com; durga@nitrkl.ac.in

Received January 4, 2010; revised November 25, 2010.

Abstract Even after thorough testing, a few bugs still remain in a program with moderate complexity. These residual bugs are randomly distributed throughout the code. We have noticed that bugs in some parts of a program cause frequent and severe failures compared to those in other parts. Then, it is necessary to take a decision about what to test more and what to test less within the testing budget. It is possible to prioritize the methods and classes of an object-oriented program according to their potential to cause failures. For this, we propose a program metric called *influence metric* to find the influence of a program element on the source code. First, we represent the source code into an intermediate graph called *extended system dependence graph*. Then, forward slicing is applied on a node of the graph to get the influence of that node. The influence metric for a method m in a program shows the number of statements of the program which directly or indirectly use the result produced by method m. We compute the influence metric for a class c based on the influence metric of all its methods. As influence metric is computed statically, it does not show the expected behavior of a class at run time. It is already known that faults in highly executed parts tend to more failures. Therefore, we have considered *operational profile* to find the average execution time of a class in a system. Then, classes are prioritized in the source code based on *influence metric* and *average execution time*. The priority of an element indicates the potential of the element to cause failures. Once all program elements have been prioritized, the testing effort can be apportioned so that the elements causing frequent failures will be tested thoroughly. We have conducted experiments for two well-known case studies — *Library Management System* and *Trading Automation System* — and successfully identified critical elements in the source code of each case study. We have also conducted experiments to compare our scheme with a related scheme. The experimental studies justify that our approach is more accurate than the existing ones in exposing critical elements at the implementation level.

Keywords program slice, extended system dependence graph, influence of a class, test priority

1 Introduction

Software solutions are increasingly permeating our everyday life. Software industries are in immense pressure to provide very reliable products where tolerance to bugs is very low. As testing is an action of sampling, it is always needed to make a decision about what to test and what not to test, what to do more or to do less. Though a systematic testing can reveal bugs, but it is very much expensive. In systematic testing approaches such as white box testing and black box testing methods, the basic problem is that they generate too many test cases, out of these test cases, some are less important[1]. Further, in traditional white box testing techniques such as statement coverage and branch coverage, each element① (method) of the software product is tested with equal thoroughness[2]. Due to the vast number of possible test cases, it is not always possible to rigorously test each and every part of a system under test within the testing budget. It is observed that some residual bugs are left even after thorough testing. These residual bugs are randomly distributed throughout the source code. The presence of bugs in some parts of the source code may cause severe and frequent failures compared to other parts. For example, if a method of a class produces crucial data that is used by many other classes, then a bug in this method would affect many other classes. Upon analyzing the software failure history data from nine large IBM software products, Adams[3] demonstrated that a relatively

Regular Paper

This work is supported by grants from the Department of Science and Technology, Government of India under SERC Project.

①A method is considered as an element in our approach.

small proportion of bugs in the software account for the vast majority of reported failures in the field. The empirical reports like [4] also propose a Pareto distribution such as 80% of all defects are found in 20% of the modules. Identifying the critical parts② in a software system remains a major question. Our goal in this paper is to identify the critical elements in the source code. Once the critical elements are identified, more exhaustive testing has to be carried out to minimize bugs in those elements. This means we can cut down testing in less critical elements. It could help to minimize the post-release failures and make the testing process more effective. As a result the user's perception on system reliability will be increased, without increasing the testing resources. User's perception is a good indicator on the acceptance of a system. Please note that we use the terms bug, fault and defect interchangeably when no confusion arises. An error done by a programmer results in a defect (fault, bug) in the software source code. The execution of a defect may cause one or more failures. As per the IEEE standard, failure is the inability of a system or component to perform its required functions within specified performance requirements. A failure in a system can be observed by the user externally. Failure is neither a coding fault in the development process nor a defect that puts an element of the program or the system into an erroneous state that does not lead to a visible failure.

The user's view on the reliability of a system is improved, when faults which occur in the most frequently used parts of the software are almost removed[5-8]. Removing faults from frequently executed parts of the source code helps the test manager to achieve high reliability in low cost. However, the length of time a part of the source code is executed does not wholly determine the importance of the part in the perceived reliability of the system. It is possible that the result produced by a function which is executed only for a small duration is saved and extensively used by many other functions. Sometimes, the produced result of a rarely executed function is widely used by a number of frequently executed functions. Hence, a function would have a very high impact on the reliability of the system even though it is itself getting executed only for a small duration. This concept motivates us to consider the interlink with other elements through coupling in order to prioritize elements in the source code. The degree of coupling can be correlated with the criticality of a system. Foyen *et al.*[9] have defined dynamic coupling measures (import coupling and export coupling) in the context of object-oriented design and the code analysis. The criticality of a class in a system is decided by measuring the *export coupling* of that class[10]. The export coupling for a class c (exports services) is defined as the number of other classes that are directly using the data members or results produced by the member functions of the class c. Export coupling is used as a measure for criticality of a class because defects in a class that exports services are particularly critical as they may propagate easily to other parts of the system and are difficult to isolate[10]. Faults in objects with higher Import Object Coupling③ could easily manifest themselves into failures of the applications because services of these objects are required by other objects[11]. A class which provides services to others is more reusable because it is more independent whereas a class importing services may be difficult to reuse in another context because it depends on many other classes. Coupling is also related to change proneness[12]. The authors of [9] also experimentally proved that a class that provides services to many number of classes is more likely to change. It is because it has to adjust to the evolving needs of many dependent classes. So, more testing efforts should be assigned to the classes which are providing services to others because bugs in those are more infected to other parts of the source code and those classes are also more reusable and more changeable④ . However, in our work, we are focusing the dependency nature of a class. The changeability attribute is basically considered at the regression testing.

Assuming that all classes are approximately of similar size and complexity, the failure rate of a software product would be disproportionately influenced by the presence or absence of bugs in some classes. These classes either get executed frequently than others during the normal operation of the software or the results produced by these classes are used extensively by a large number of other classes. Hence, we define the criticality of a class based on its two characteristics such as execution probability and coupling. The first one is determined by considering the *operational profile*[6] prepared for the system at an early stage and the later one by the help of *coupling*[10]. In this work, we present a new metric called *influence metric* for an

②A part is critical if it is responsible for frequent failures.

③The import and export coupling defined in [11] is differ in the direction of coupling (import and export) defined in [10]. The authors of [11] stated that export coupling between two objects (o_1, o_2) is based on how many messages the object o_1 is sending to object o_2. According to the works in [9-10], an object providing how much services to others is a measure of export coupling of that object whereas it is stated as import coupling in [11] as these services are imported by others.

④The class which is depending on more number of classes is also more changeable due to change in determinate classes.

element e. It shows how many elements in a program are using the produced result of e directly or indirectly. Our proposed influence metric provides detailed information at statement level. It shows how many portions of the source code are influenced by an element. The *influence_value* derived from influence metric of an element e is the measure of criticality of e. As the analysis is performed in the code level, our proposed procedure marks the nodes (statements) in the source code that are depending on e directly or indirectly. First, we propose an algorithm to compute the influence of a method and then we use it to compute the influence of a class. To identify the criticality of an element e in a program, we assign test priority to e based on its *influence_value* (which shows how many nodes are dependent on it) and *average execution time* (which shows how often these dependencies are executed at run time).

The rest of the paper is organized as follows. The basic concepts, which are used later in this paper are described in Section 2. Our proposed influence metric for an object-oriented program and the assignment of test priority to a class is discussed in Section 3. In Section 4, we present the experimental studies. We review the related work in Section 5 and conclude the paper in Section 6.

2 Basic Concepts

In this section, we present certain basic concepts and terminologies associated with our work that are used in later sections. We first present the basic idea of slicing and then intermediate representation of programs.

2.1 Program Slice

A program slice is a part of the code that contributes in computation of certain variables at a program point of interest. For a statement s and variable v, the slice of a program P with respect to the slicing criterion $\langle s, v\rangle$ includes only those statements of P that are needed to capture the behavior of v at s[13]. Slicing can be static or dynamic. Static slicing technique uses static analysis to derive slicing. That is, the source code of the program is analyzed and the slices are computed for all possible input values. No assumptions are made about input values. Since the predicates may be evaluated as either true or false for different values, conservative assumptions have to be made, which may lead to relatively large slices. So a static slice may contain statements that might not be executed during an actual run of a program, whereas dynamic slicing makes use of the information about a particular execution of a program. The execution of a program is monitored and the dynamic slices are computed with respect to execution history. A dynamic slice with respect to a slicing criterion $\langle s, v\rangle$, for a particular execution, contains only those statements that actually affect the slicing criterion in the particular execution. Therefore, dynamic slices are usually smaller than static slices and are more useful in interactive applications such as program debugging and testing.

Forward slice with respect to a slicing criterion $\langle s, v\rangle$ contains all the parts of the program that might be affected by the variable in v used or defined at the program point s. A forward slice provides the answer to the question: "which statements will be affected by the slicing criterion?"[14].

2.2 Program Representation

In the following, we present a few basic concepts associated with intermediate representations of programs that are used in later sections.

2.2.1 Program Dependence Graph (PDG)

The program dependence graph[15] G of a program P is the graph $G = (N, E)$, where each node $n \in N$ represents a statement of the program P. The graph contains two kinds of directed edges: control dependence edges and data dependence edges. A control (or data) dependence edge (m, n) indicates that n is control (or data) dependent on m. Note that the PDG (Program Dependence Graph) of a program P is the union of a pair of graphs: data dependence graph and control flow graph of P.

2.2.2 System Dependence Graph (SDG)

PDG cannot handle procedure calls. Horwitz *et al.*[16] introduced the System Dependence Graph (SDG) representation which models the main program together with all associated procedures. SDG is very similar to PDG. Indeed, a PDG of the main program is a subgraph of SDG. In other words, for a program without procedure calls, PDG and SDG are identical. The technique for constructing an SDG consists of first constructing a PDG for every procedure, including the main procedure, and then adding dependence edges which link various subgraphs together.

An SDG includes several types of nodes to model procedure calls and parameter passing:

• Call-site nodes represent the procedure call statements in the program.

• Actual-in and actual-out nodes represent the input and output parameters at call site. They are control dependent on the call-site nodes.

• Formal-in and formal-out nodes represent the input and output parameters at called procedures. They

are control dependent on procedure's entry node.

Control dependence edges and data dependence edges are used to link an individual PDG in an SDG. The additional edges used to link a PDG are as follows.

• Call edges link the call-site nodes with the procedure entry nodes.

• Parameter-in edges link the actual-in nodes with the formal-in nodes.

• Parameter-out edges link the formal-out nodes with the actual-out nodes.

• Summary edges connects an actual-in vertex and an actual-out vertex if the value associated with the actual-in vertex may affect the value in actual-out vertex. It represents the transitive dependencies that arise due to procedure calls.

2.2.3 Extended System Dependence Graph

Extended System Dependence Graph (ESDG) models the main program with all other methods. Each class in a given program is represented by a class dependence graph. Each method in a class dependence graph is represented by procedure dependence graph. Each method has method-entry vertex that represents the entry in the method. The class dependence graph contains a class-entry vertex that is connected with the method-entry vertex of each method in the class by a special edge known as class-member edge. To model parameter passing, the class dependence graph associates each method-entry vertex with formal-in and formal-out vertices.

The class dependence graph uses a call vertex to represent a method call. At each call vertex, there are actual-in and actual-out vertices to match with the formal-in and formal-out vertices present at the entry to the called method. If an actual-in vertex affects an actual-out vertex then a summary edge is added at the call-site, from the actual-in vertex to the actual-out vertex to represent a transitive dependency. To represent inheritance, we construct representations for each new method defined by the derived class, and reuse the representations of all other methods that are inherited from the base class. To represent the polymorphic method call, ESDG uses a polymorphic vertex. A polymorphic vertex represents the dynamic choice among the possible destinations. The detailed procedure for constructing an ESDG can be found in [17]. Each node can be a simple statement or a call statement or a class-entry or a method-entry vertex.

Several researchers[18-20] have proposed different types of intermediate representation for object-oriented software. Rothermel and Harrold[18] extended PDG, and proposed the Class Dependence Graph (ClDG) for use in regression testing. Larsen and Harrold[17] extended SDG by representing a class with a ClDG, and proposed a system dependence graph for object-oriented software (ESDG). The basic aim of designing ESDG is to get a slice of an object-oriented program on the basis of graph reachability. Liang and Harrold[20] proposed extensions to ESDG for the purpose of object-slicing. Malloy *et al.*[19] also proposed a layered representation, the Object-oriented Program Dependency Graph (OPDG), by adapting the basic concepts of PDG. Out of these, we are considering ESDG by Larsen and Harrold[17] because our main aim is to get a forward slice of a method-entry vertex through the process of graph reachability.

In rest of the paper, we use the terms node and vertex interchangeably.

3 Overview of Our Approach

An object-oriented program comprises of a set of classes. To compute the influence for a class, we first construct an intermediate representation called *ESDG*[17] through the static analysis of the program code. Then, forward slice on each method-entry vertex v of a class is determined based on graph reachability algorithm[13]. This slicing technique marks the nodes of ESDG that are depending on v directly or indirectly. First we compute the influence of a method in a class. Combination of influence of all relevant methods of a class c is the influence of the class c. This approach statically computes the influence of a class in a program. Execution of the program is not necessary. Though, the influence set of a class c (derived through our proposed method) shows all possible requests to c for service, but it is unable to show how often these requests are executed in the operational environment.

The reliability of a system is not related to the number of existing faults in a system under test, but related to the probability that a fault leads to a failure that occurs during software execution[21]. It is because the data input supplied by the user decides which parts of the source code will be executed. An error existing in the non-executed parts will not affect the output. So, it is not only sufficient for a class c to know how many other classes are requesting services from c, but also how often these requests are executed at run time. For this, we are extracting the average execution time of a class based on *operational profile*[6] of the system under test. The operational profile is the probability with which different high-level functions (or use cases) are executed during a typical use of a software. We assume that the operational profile is already decided at the requirement stage and how to get the operational profile

of a system is out of the scope of this paper. The reader can refer to [6, 22-24] for details on operational profile. Once both the factors of a class: *influence_value* and *average execution time* are computed, we can assign test priority to each class based on these two factors. Test priority value assigned to a class reflects its level of criticality in a program. Hence, test priority assigned to different classes in a program shows their intensity of testing requirements. The higher the test priority of a class, the more the testing effort assigned to test it thoroughly.

3.1 Influence of a Method

In a program, the incorrect results computed by a called method may affect other calling methods. It is because, the value returned by the called method may be used by a calling method, for taking any further action or for some computation, which inaccuracy may lead to catastrophic consequence. The influence of a given method M is defined by the number of other statements of the given program, those are directly or indirectly using the result computed by it. If the influence level of a method is higher, then that method is more critical. First, we represent the input program as an intermediate representation (ESDG). Then, we apply our proposed algorithm on the ESDG to compute the influence of a method in a program. We count the number of nodes marked as influenced by a method M in a program from the data dependent set of that method's formal parameter-out nodes.

The influence of a method M is expressed as:

$$\frac{\text{No. nodes influenced in ESDG by } M}{\text{Total No. nodes in ESDG}}. \quad (1)$$

In this subsection, we present our algorithm *MethodInfluence* for computing the *influence_value* of a method in pseudo code form in Fig.1. The notations used in our algorithm is presented below.

• *visited*[i]: it is a Boolean variable which is set to TRUE upon visiting node i.

• *influence*[i]: it is a Boolean variable which is set to TRUE when node i is marked as influenced.

• *queue*1: it is a queue that contains the nodes which are to be processed next.

• *queue*2: it is a queue that contains the nodes which are to be marked as influenced.

• *insertQueue*: it is a function that inserts data to a queue.

• *deleteQueue*: it is a function that deletes data from a queue.

• *Type*(n): it is a function that returns the type of node n out of all possible types in ESDG.

Algorithm 1. MethodInfluence (*ESDG*, V_{me})

Require: *ESDG*: intermediate representation of the program, N: total number of nodes in *ESDG*, V_{me}: method-entry vertex of the method M.

Ensure: $Inf(M)$: *influence_value* of the method M.

```
1:  for i ← 1 to N do
2:    visited[i] ← FALSE
3:    influence[i] ← FALSE
4:  end for
5:  Queue queue1 ← ∅
6:  Queue queue2 ← ∅
7:  insertQueue(queue1, V_me)
8:  while queue1 ≠ ∅ do
9:    n ← deleteQueue(queue1)
10:   if Type(n) == method-entry vertex then
11:     Traverse only its control-edges and parameter-edges
12:   else
13:     if Type(n) == call vertex then
14:       Traverse its all types of edges
15:     end if
16:   else
17:     if Type(n) == polymorphic vertex then
18:       Traverse only its polymorphic-edges
          {each adjacent node of a polymorphic-edge is a
           method-entry vertex}
19:     end if
20:   else
21:     Traverse only its outgoing data-edges and control-
        edges
22:   end if
23:   for each adjacent not-visited node w do
24:     visited[w] ← TRUE
25:     if Type(w) == parameter-out vertex then
26:       insertQueue(queue2, w)
27:     else
28:       insertQueue(queue1, w)
29:     end if
30:   end for
31: end while
32: while queue2 ≠ ∅ do
33:   n ← deleteQueue(queue2).
34:   influence[n] ← TRUE
35:   Traverse through all types of edges of node n except
      the control-edges
36:   for each not-visited node w do
37:     visited[w] ← TRUE
38:     insertQueue(queue2, w)
39:   end for
40: end while
41: Calculate Inf(M) using (1)
42: return Inf(M)
```

Fig.1. MethodInfluence algorithm.

3.2 Working of the Algorithm

To illustrate how to compute the static influence of a method in a class, we consider the example program of Fig.2 and its ESDG shown in Fig.3. Consider the method *add* of class *Task* in the program given in Fig.2. Now our proposed algorithm starts from the given method-entry vertex 2. We traverse each control edge from the given method-entry vertex and enqueue the nodes $F1\_in$, $F2\_in$, 3 in *queue*1 and F_out in *queue*2. Now it will dequeue the first element $F1\_in$ and checks all its outgoing edges to find any depending node that is not traversed. Again it will dequeue the nodes $F2\_in$ and 3 as long as there are no other nodes, *queue*1 becomes empty.

```
Class Task {                          9:   while (i < 11) {
      public:                         10:  sum = add(sum, i);
2:    int add(int x, int y){          11:  i = incr(i); }
3:    return (x + y); }               12:  cout << "SUM = " << sum;
4:    int incr(int i){                    }
5:    i = i + 1;}                     main () {
6:    void fun() {                          Task ob;
7:    int sum = 0;                    1:    ob.fun();
8:    int i = 1;                            }
```

Fig.2. A sample object-oriented program.

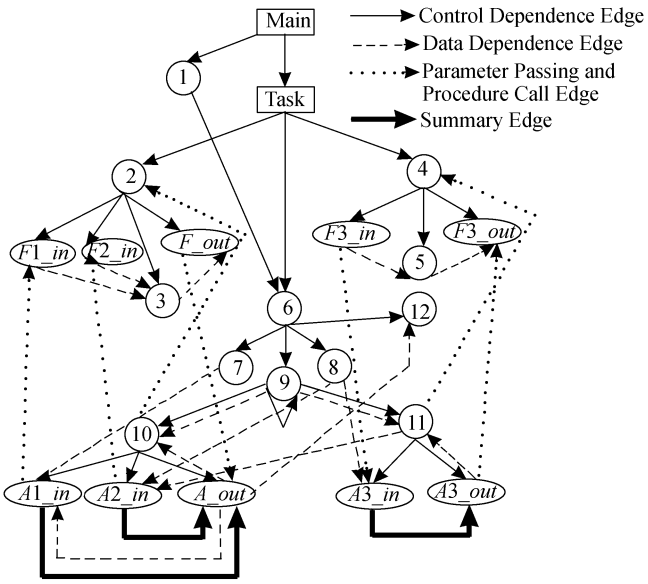


Fig.3. Extended system dependence graph of Fig.2.

After that it will dequeue F_out from *queue*2. Mark it as influenced and then, traverse all its *parameter-out* edges only. Enqueue all the not-visited nodes in *queue*2. Now *queue*2 contains A_out. Dequeuing A_out and marking it as influenced, next *queue*2 will contain 10, $A1\_in$ and 12. In this similar way, nodes are inserted in *queue*2 and deleted from it. At the end, we will get the nodes F_out, A_out, $A1\_in$, 10 and 12 are marked as influenced. It shows the contribution of *add* method to rest of the source code.

3.3 Influence of a Class

The nodes in the set $Influence\_of\_class(c)$ for the class c is the union of all the sets $Influence\_of\_method(m_i)$, where m_i is the i-th method of class c.

$$Influence\_of\_class(c) = \bigcup_{i=1}^{k} Influence\_of\_method(m_i),$$

where k is the total number of methods in class c.

3.4 Complexity Analysis

If N number of nodes are created in the intermediate graph (ESDG) for representing the object-oriented program, at each node there can be maximum $N-1$ number of edges. So, worst case space complexity will be $N \times (N-1) = O(N^2)$. Similarly, in the ESDG any edge is visited at most once. So, time complexity $= O(m)$, where m is the total number of edges.

3.5 Average Execution Time of a Class ($ET\ (C_i)$) in a System

An operational profile assigns probability values to different use cases[⑤] based on probability of use of the high level functions (use cases) by different user types. Suppose we have drawn a use case diagram consisting of m types of users and n number of use cases for a practical application. Each user type has assigned a probability of using the application system. Let u_i be the probability assigned to i-th user type of accessing the application such that

$$\sum_{i=1}^{m} u_i = 1.$$

Let q_{ij} be the probability of requesting the functionality of j-th use case ($j = 1 \ldots n$) by i-th type user ($i = 1..m$) such that

$$\sum_{j=1}^{n} q_{ij} = 1.$$

The probability value of a use case x denotes the likelihood of the use case being executed by an average user

⑤We identify the use cases with high-level functional requirements of the system.

and is expressed as:

$$p(x) = \sum_{j=1}^{m} u_j \times q_{jx}.$$

We consider that the functionality of any system can be modeled through a set of scenarios derived from use cases[25]. From each use case, a number of scenarios are identified by drawing the sequence diagram for each possible use of use case. Now, we assign non uniform probability distribution to each scenario based on its frequency of execution. As per the domain knowledge, each scenario of a use case is assigned some frequency value based on a number of executions in a particular environment for a particular time period. Let $f_i(j)$ be the frequency of j-th scenario of i-th use case such that

$$\sum_{j=1}^{nos_i} f_i(j) = 1,$$

where nos_i is the total number of scenarios of i-th use case. Then, the probability of execution of k-th scenario of i-th use case is $p(k_i) = p(i) \times f_i(k)$.

Each scenario is implemented by the interaction among a set of classes. The average execution time of a class $ET(C_i)$ in the system is given by:

$$ET(C_i) = \sum_{j=1}^{nos} p_j \times (Time(C_i)),$$

where nos is the total number of scenarios in a system under test, p_j is the probability of j-th scenario and $Time(C_i)$ is the total activation time of C_i in j-th scenario.

3.6 Test Priority Calculation

Testing effort can be assigned to a class based on its criticalness in a program. We combine both *influence_value* and *average execution time* of a class to get the *Test Priority* (*TP*) for that class. *TP* for a class is computed by applying the following formula.

$$TP(c_i) = Inf\_val(c_i) \times ET(c_i), \quad (2)$$

where $inf\_val(c_i)$ and $ET(c_i)$ are the *influence_value* and *average execution time* for class c_i. The degree of criticalness for a class is decided based on its *TP* value (a class having high *TP* value is more critical). *TP* for different classes of a small program (here, the *Book Management* module of a Library Management System is considered. In Section 4, Library Management System is discussed as a case study) is computed using (2) and is shown in Table 1.

Table 1. Test Priority (*TP*) Calculation

C_i	*Inf_val* (%)	$ET(c_i)$	Total *TP*	*TP* (%)
1	22	55	1210	22
2	12	75	900	16
3	15	38	570	11
4	45	58	2610	49
5	6	15	90	2
Sum	100	–	5380	100

3.7 Limitation of Our Approach

For a better result, it is not sufficient to assign testing effort to an element based on its test priority value as computed in this work. Sometimes, it is found that a single bug in a class having lowest priority value could cause catastrophic failure. For example, the failure of an exception handler causes a sever loss to the system. As, some classes usually provide exception handling of rare but critical conditions, it is necessary to consider the severity associated with each class by checking the effect of its failure to the system operation. Severity analysis[26] is beyond the scope of this paper. So, for efficient testing, the test priority computation should also include the severity associated with the failure of a class. The limitation of our approach is that we are not considering the severity associated with each class. We are planning to include it in our future work. Another limitation is that though ESDG is good for representing small and moderate programs, but for a large real life program, ESDG may be too large to manage[27]. Obviously, the storage management will also be very high. For large programs, influence metric will be computed by using traditional *fan-in* and *fan-out* metric[28] in place of ESDG. However, the advantage of using ESDG over local fan-in and fan-out is that our proposed influence metric will be more accurate as ESDG shows the details regarding the statements that are really affected in source code, when a method is producing incorrect result. It is because ESDG shows dependencies at statement level, fan-in and fan-out shows higher level dependencies at the function level.

4 Experimental Studies

We have implemented our proposed algorithm for the calculation of influence metric for simple JAVA programs. The intermediate graph used in the algorithm is obtained using ANTLR in the ECLIPSE framework. We have considered two case studies — Library Management System (LMS) and Trading-house Automation System (TAS). These two case studies are selected from the programming assignments done by the students of our institute. These case studies are neither very small nor very large, but of moderate size. The case studies

are well explained in [2]. We present a brief summary of these case studies in Table 2, so that the size of each can be well understood. In this table, *object points* shown in Column 4 are estimated based on how many individual *screens* are displayed, how many *reports* are produced and number of *3GL modules* developed in the system[29-30]. The value of No. Cl shown in Column 3 of Table 2 are user classes. System classes are not considered here.

Table 2. Features of the Case Studies

Case Study	No. UC	No. Sc	No. Cl	No. OP	LOC
LMS	14	56	16	153	2850
TAS	13	68	12	25	2140

Note: UC: Use Cases, Sc: Scenarios, Cl: Classes, OP: Object Points, LOC: Line of Code

4.1 Sensitivity Analysis

Using our test priority estimation, we investigate the failure rate of an application based on the failure of individual classes. We have done it in three phases. In the first phase, we selected the highest priority class from a case study and decreased its reliability⑥ , while fixing the reliabilities of other classes to 1.0, for the sake of comparison. To observe the failure rate of the application, we select randomly 100 numbers of test cases (randomly selected scenarios) based on operational profile. A test case is responsible for the execution of one scenario⑦. Then, we continue our process by slowly decreasing the reliability of a selected class in a step wise manner and observe the failure rate of the system under test at each reliability point of that class for the same set of test cases. Same process and same test cases are also applied to a class having medium priority and to a class having the lowest priority. As failures are observed by executing the same set of test cases at each reliability point of selected classes (one at a time), it helps to analyze the sensitivity of a class towards failures. The graphs in Fig.4 show the failure rates of LMS and TAS case studies by decreasing the reliability of the highest priority class, some medium priority classes and the lowest priority class (one at a time) of each case study. We have considered six classes of each case study including the highest and lowest priority class.

Let us consider the LMS case study. In Fig.4, it is clearly shown that when the reliability decreases for the class having the highest test priority, system failure rate increases at a higher rate, but this is not true for

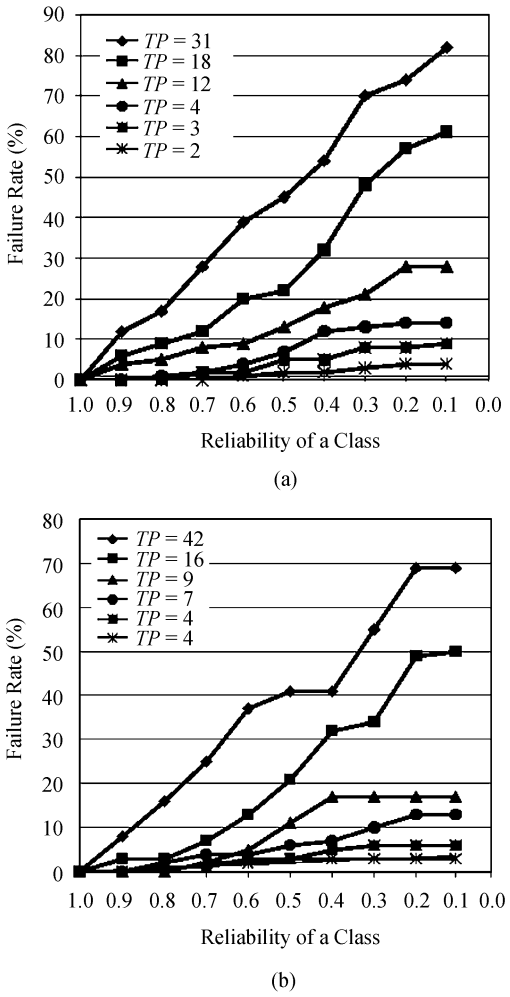


Fig.4. Failure rate of the application based on class reliabilities (one at a time). (a) Sensitivity analysis of LMS. (b) Sensitivity analysis of TAS.

low test priority classes. Now, we explain the effect of one such seeded bug in a medium priority class *Borrower* (bugs are muted in a class to decrease its reliability). Suppose a borrower wants to issue a book. As per the business logic of LMS, a borrower can issue only one book. So, after issuing a book, the *Borrower* object should be changed from active state to blocked state, but the seeded bug in the issue() method of *Borrower* class has skipped this. As a result a failure is not observed in current transaction but will be observed in future. It is because the system is violating the specification by allowing the same borrower to issue another book. The system is also allowing the borrower to be deleted which is also violating the specification. The

⑥Techniques for class reliability estimation is a step wise procedure that includes fault injection, testing and retrospective analysis. How to assess the reliability of a class is outside the scope of our discussion. We are assuming an estimate is available, this is used as a parameter for observing the failure rate to analyze the sensitivity of the application.

⑦A scenario may be executed more than once for different test values.

failure rate for the system will be increased drastically, if the same bug is seeded in class *Book* instead of class *Borrower* that will not change the state of the object *Book*. In this case, as the book is now in available state though it is already issued, frequent failures will be observed in book transactions and daily reports.

4.2 Comparison with Musa's Approach

We have argued that failure-prone classes are identified not only by the execution time but also by the influence metric. Like *average execution time*, a class having high *influence_value* is also responsible for a high failure rate of the overall system. To validate our claim, we have conducted two experiments on each case study (LMS, TAS). In Experiment 1, we checked the impact of *execution time* on system failure rate and in Experiment 2, the impact of *influence metric* is checked.

Experiment 1 (Extended Musa's Approach). We have extended the existing Musa's approach[6] to class level, for sensitivity analysis. For each case study (LMS, TAS), first five classes are taken from a set of classes arranged in descending order according to their *average execution time*. In this experiment, we have ignored the *influence_value* of a class. Then, we applied the same technique and the same dataset as discussed in Subsection 4.1 for sensitivity analysis, to the selected five classes.

Experiment 2 (Checking the Impact of Newly Introduced Factor: *Influence_value*). In this experiment, we have to prove that a rarely executed class c is also responsible for increasing the system failure rate, if c is providing services to a number of classes in a system. It may happen that the produced result of a class is saved and used by a number of classes. Hence, we check the impact of *influence_value* of a class on system failure rate. For each case study (LMS, TAS), five classes are selected based on low *average execution time* and high *influence_value* for sensitivity analysis. We have applied the same technique and same dataset as in Experiment 1. Then, we checked the tendency of each selected class towards overall failures of the application. For simplicity, we have considered only five classes in both the experiments. It may vary according to a project's size. The procedure to select the required number of classes for this experiment is as follows.

(i) A graph is drawn for all classes of a case study taking the *influence_value* in decreasing order along X-axis and *average execution time* in increasing order along Y-axis.

(ii) A forward movement is done slowly in both axes from the origin with the intention of selecting points near the origin. The movement continues until the target number of points (classes) are collected.

This can be better explained through a small example. Suppose, we have to select two classes according to the given criteria from Table 1. First *influence_value vs. execution time* graph for each class of Table 1 is drawn as shown in Fig.5. As our aim is to select points near the origin, we move slowly in both the directions. We check any class lying in the origin (100, 0). As there is no class in this point, our next search point is (90, 10). This way we search (80, 20), (70, 30).... A class having *influence_value* $\geqslant x$ and *average execution time* $\leqslant y$ will lie in a point (x, y). We get the class C_4 at the point (40, 60) and then class C_1 at the point (20, 80). So, the two classes selected from Table 1 based on high *influence_value* and low *average execution time* are C_4 and C_1. The other classes of the said table are not selected due to having either low *influence_value* or high *average execution time*.

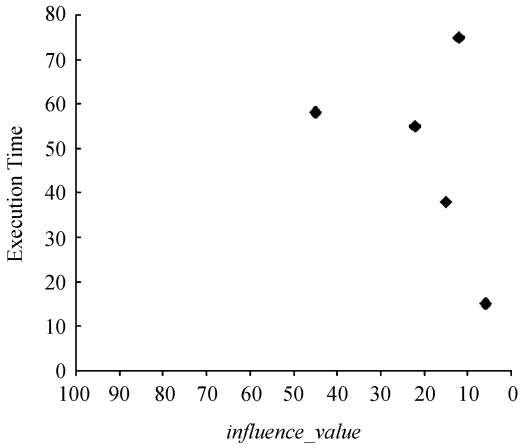


Fig.5. *influence_value* vs. *average execution time*.

In the first experiment, for LMS and TAS case studies, the maximum system failure rate of the overall system was 58% and 47% respectively, when the reliability of the first class (the class having the highest execution rate) was decreased from 1 to 0.5. In the second experiment, we found that the maximum system failure rate was near about 45% and 46% for LMS and TAS respectively, when the reliability of some classes out of the selected five classes (one at a time) decreased from 1 to 0.5.

From Experiment 1, it is observed that the failure rate is increased, when the reliability is decreased for a class having high *average execution time*. From Experiment 2, it is observed that a class having high *influence_value* is also responsible for increasing the failure rate. Through the experimental studies in this subsection, we found that the newly introduced factor *influence_value* is also nearly as competent

as *average execution time* in identifying failure-prone classes. Hence, we conclude that *influence_value* is also playing a major role in identifying the failure-prone classes, whereas Musa[6] stated that the frequently executed classes should get more focus on the testing phase as they are more failure-prone. As our proposed prioritization method is ranking classes based on *average execution time* and *influence_value* (A class having high *execution time* and high *influence_value* is getting high rank), it exposes the failure-prone classes that are exposed by Musa's approach[6]. Further, our method identifies some more failure-prone classes through the newly introduced factor *influence_value* that are neglected by Musa's approach due to low execution time. It may happens that the wrongly produced output by a rarely executed class is used by some frequently executed classes, that is causing the system failure rate high. It can be better explained through the graphical representation of a simple example instead of going to the details of the case studies. Consider the sequence diagrams shown in Fig.6. Suppose the execution probability of $SD1$ and $SD2$ is 80% and 20%. Now, the average execution time of class D is the lowest, but the influence is more as it is providing services to more number of classes. As a result, the failure rate of the system will be high. It is because a fault in class D may affect the highly executed classes A, B and C.

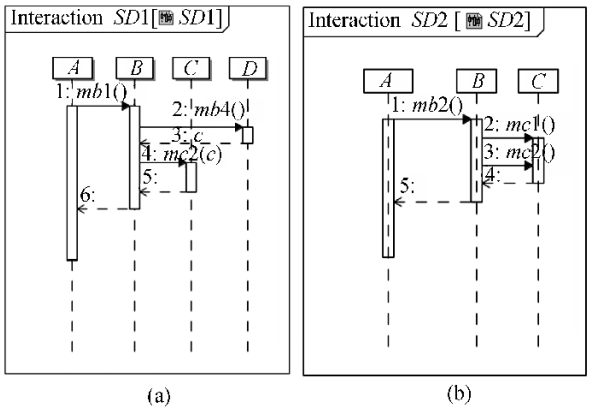


Fig.6. Two sample sequence diagrams. (a) $SD1$. (b) $SD2$.

4.3 Threats to Validity of Results

In order to justify the validity of the results of our experimental studies, we identified the following list of threats:

• Biased test set design and influencing results.

• Seeding biased errors in different classes of each case study.

• Testing only for selected failures and loosing generality of results.

• Using testing methods which may only be suitable for some particular bugs while may not reveal other common and frequent bugs.

In order to overcome the above mentioned threats and validate the results for most common and real life cases, we have taken the following corrective measures:

• For each discussed case study, we have used same test set at each reliability point of a class for observing failures and applied the same test set for each class.

• We have used same type of seeded bugs in each class of each case study (one class at a time) by different groups of students. There were no interaction between the two groups at the time of bug seeding.

• We have taken care that the seeded bugs match with commonly occurring bugs.

• We have inserted some class mutation operators to seed bugs. Using mutation operators, we can ensure that a wide variety of faults are systematically inserted in a somewhat impartial and random fashion. While traditional mutation operators are restricted to a unit level, class mutation operators[31] for object-oriented programs have impact on cluster level.

• We are considering all failures that provide a base to the user to decide how much they can trust the software.

5 Related Work

Researchers have proposed a variety of prioritization-based testing techniques in order to do the best possible job with limited resources. These techniques make the testing process more effective and cheaper. An effective test could find more number of defects or more important defects in the same amount of time. As testing is expensive and time consuming, the test manager has to choose among alternatives, not use them all. A lot of work has been done on *Test Case Prioritization* but, work on *Source Code Prioritization* has scarcely been reported. It is a research area on improving testing before test case generation. First, we discuss the work on *Source Code Prioritization* and compare it with our proposed work. Then, we discuss on another related research area on pre-testing efforts with same objective such as *Usage Based Testing*. Lastly, we present a review on the reported work on some *Test Case Prioritization* techniques.

5.1 Code Prioritization

The basic aim of this technique is to focus on testing efforts before the generation of test cases. It helps to prioritize the testing efforts based on test objectives. With a prioritized testing effort and focused test architecture, test cases are created and executed.

Li[32] has proposed a priority calculation method that prioritizes and highlights the important parts of the source code based on *dominator analysis*, that need to be tested first to quickly improve the code coverage. Before test construction, Li's method decides which line of code will be tested first to quickly improve code coverage. According to his approach, first the intermediate representation of the source code, known as Control Flow Graph (CFG) is constructed. Then, a node⑧ of CFG is prioritized based on measuring quantitatively how many lines of code are covered by testing that node. A weight is calculated for each node considering only the coverage information. It does not take into account, for instance, the complexity or the criticality of a given part of the program. A test case covering the highest weight node will increase the coverage faster⑨. There are two kinds of code coverage such as control flow based and data flow based. Li's work focuses on control flow coverage. Code coverage helps the developers and vendors to indicate the confidence level in the readiness of their software, but the limitation is that it gives equal importance to the discovery of each fault. So, no information is gained on how much it affects the reliability of a system by detecting and eliminating a failure during testing process, as different failures have different contribution to the reliability of a system. Though Li's work[32] is more effective at finding bugs, but it very often spends more resources by uncovering many failures having occurrence rate very much negligible during actual operation. As a result of this, sometimes test efforts are wasted without appreciably improving the reliability of the software. Therefore, this code prioritization technique could not always detect the bugs, which are responsible for frequent failures, when the system is executed for some duration by providing some random inputs. Unlike Li's work[32] on code prioritization, our code prioritization strategy analyzes failure-prone parts in the source code and is able to focus first on detecting the faults that cause the most frequent failures. The failure-prone parts are identified based on influence metric and operational profile. We compare our work with the existing work on code prioritization[32] according to the following six criteria given in Table 3.

Li *et al.*[33] presented a methodology for code coverage-based path selection and test data generation, based on Li's previous work[32]. They[33] proposed a path selection technique that considers the program priority and call relationships among class methods to identify a set of paths through the code, which has high priority code unit. Then, constraint analysis method is used to find object attributes and method parameter values for generating tests to traverse through the selected sequence of paths. It helps to automatically generate tests to cover high priority points and minimize the cost of unit testing.

Table 3. Comparison of Existing Work on Code Prioritization with Our Work

Comparison Criteria	Li's Work[32]	This Work
Aim of code prioritization	Quickly improve code coverage (control flow based coverage)	Quickly improve user's perception on the reliability of the system
Priority level	Line of code or block	Method or class
Analysis point	Structural	Structural and behavioural
Factors for prioritization	Number of lines of code covered	Influence and avg. execution time
Intermediate graph used	Control flow graph	ESDG (both data and control flow)
Fault criticality	Equal importance to the discovery of each fault	Faults in high priority areas are more critical and hence given more importance.
Effectiveness	Finding more number of bugs	Finding bugs that have more contribution to unreliability

5.2 Usage Based Testing

The first step in usage based testing is to develop a usage model that describes the anticipated behavior (usage) of the system under testing. Usage model helps to improve the user's perception on the system's reliability. It is also designed before the construction of test cases. It basically represents the events and transition between events in the system, where events can be user input or environmental input. Its main purpose is to describe the possible behaviors of the user and to quantify the actual usage in terms of probabilities for different user behavior. Though unlike our approach, no prior knowledge of the program is necessary for usage based testing. The authors found a lot of research work has been done on usage based testing at the abstract level based on operational profile[6], which focuses on detecting faults that cause the most frequent failures. By prioritizing our tests based on usage probabilities, it ensures that the failures that will

⑧A node is a basic block in the source code.

⑨The tester, based on his/her experience may desire to cover first a node with a lower weight but that has a higher complexity or criticality.

occur most frequently in operational use will be found early in the test cycle, while keeping testing effort to a minimum. In terms of Mean Time Between Failure (MTBF), Cobb and Mills[7] said that "It is shown based on a study of a number of projects that usage testing improves the perceived reliability during operation 21 times greater than that using coverage testing". However, these works have concentrated on selection of test cases based on black-box approach compared to our white-box approach. As a result, it ignores the structural (syntactic and semantic) relationships that exist among the elements of source code, which is vital for reliability assessment. Sometimes the infrequently executed code of a complex module that is a source of failure could not be tested by this method. Cheung[21] has proposed a user-oriented software reliability model, which measures the quality of service that a program provides to a user. His Markov reliability model uses a program flow graph to represent the structure of the system. The flow graph structure is obtained by analyzing the code. It uses the functional modules as the basic components whose reliabilities can be independently measured. It uses branching and function-calling characteristics among the modules as the user profile so that they can be easily measured in the operational environment. Similar structural models have been proposed by Littlewood[34] and Booth[35], to analyze the failure rates of a program. Lyu *et al.*[36] have proposed a structural model for estimating the reliability of component-based programs where the software components are heterogeneous and the transfer of control between components follows a discrete time Markov process. However, in all the related techniques discussed above, the data dependency among the components has been ignored. As a result, the tester finds it very hard to describe the interdependency of software failures in detail.

5.3 Test Case Prioritization

There has been a large number of work[18,37-39] reported on prioritizing test cases. A meaningful prioritization of test cases can enhance the effectiveness of testing within the same testing effort. Some researchers[18,37-38] have proposed test case prioritization techniques to reduce the cost of regression testing based on total requirement and additional requirement coverage. Their basic aim is to improve a test suite's fault detection rate. Elbaum *et al.*[38] have added two major attributes such as test cost and fault severity to each element of the test suit. They have experimentally validated that the test suites executed based on prioritization technique always outperform unprioritized test suites in terms of fault detection rate. The authors of [39] have also proposed a test case prioritization technique for regression testing using relevant slicing. These discussed prioritization techniques are used to improve testing by selecting appropriate test cases from the pool of test cases.

Unlike these discussed test case prioritization techniques, our aim is not to increase the rate of fault detection at the time of regression testing, which is applicable during maintenance phase. Our approach identifies the critical parts of the source code at the development phase and detects more faults from the critical parts to minimize the post-failure rate without increasing the testing effort.

6 Conclusion

We have discussed a prioritization technique to identify the criticalness of an element in the source code. Our proposed technique ensures that the classes in which a single bug may cause frequent failures in the operational environment, will be identified early. First, we have proposed a metric called *influence metric* using forward slicing to compute the influence of a class. It is based on static analysis of a program. Then, the expected behavior of a class in the operational environment is considered based on *operational profile*. Test priority is assigned to each class based on its *influence_value* and *average execution time* in a system. We have experimentally proved that when the reliability of a high critical class (criticalness of a class is decided based on its test priority value) decrease, the system failure rate increases at a higher rate, whereas this is not true in case of a low critical class. So, the intensity with which each element should be tested is proportionate to its test priority value. It helps the test manager to expose the critical elements before test case generation that are getting less attention in terms of testing. Our proposed prioritization method will be more effective, if the severity associated with each failure will be considered. We are planning to consider the severity associated with each element in our future work.

References

[1] Petschenik N H. Practical priorities in system testing. *IEEE Softw.*, 1985, 2(5): 18-23.

[2] Mall R. Fundamentals of Software Engineering, Third Ed. Prentice Hall, India, 2009.

[3] Adams E N. Optimizing preventive service of software products. *IBM Journal for Research and Development*, 1984, 28(1): 3-14.

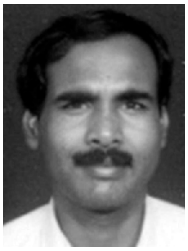
[4] Boehm B, Basili V R. Software defect reduction top 10 list. *Computer*, 2001, 34(1): 135-137.

[5] Sommerville I. Software Engineering, 5th Ed. Chapter 18, Pearson, 1995.

[6] Musa J D. Operational profiles in software-reliability engineering. *IEEE Softw.*, 1993, 10(2): 14-32.
[7] Cobb R H, Mills H D. Engineering software under statistical quality control. *IEEE Softw.*, 1990, 7(6): 44-54.
[8] Musa J D. Software Reliability Engineering: More Reliable Software Faster and Cheaper. AuthorHouse, 2004.
[9] Foyen A, Arisholm E, Briand L C. Dynamic coupling measurement for object-oriented software. *IEEE Trans. Softw. Eng.*, 2004, 30(8): 491-506.
[10] Briand L C, Daly J W, Wust J K. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 1999, 25(1): 91-121.
[11] Yacoub S M, Ammar H H, Robinson T. Dynamic metrics for object-oriented designs. In *Proc. the 6th International Symposium on Software Metrics* (*METRICS 1999*), Boca Raton, USA, Nov. 4-6, 1999, pp.50-61.
[12] Briand L C, Wuest J, Lounis H. Using coupling measurement for impact analysis in object-oriented systems. In *Proc. the IEEE International Conference on Software Maintenance*, Oxford, UK, Aug. 30-Sept. 3, 1999, pp.475-482.
[13] Weiser M. Program slicing. *IEEE Trans Software Eng.*, July 1984, 10(4): 352-357.
[14] Srikant Y N, Shankar P (eds.). The Compiler Design Handbook: Optimizations and Machine Code Generation. CRC Press, 2002.
[15] Ferrante J, Ottenstein K J, Warren J D. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 1987, 9(3): 319-349.
[16] Horwitz S, Reps T, Binkley D. Interprocedural slicing using dependence graphs. In *Proc. the ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI 1988*), Atlanta, USA, Jun. 22-24, 1988, pp.35-46.
[17] Larsen L, Harrold M J. Slicing object-oriented software. In *Proc. the 18th International Conference on Software Engineering* (*ICSE 1996*), Berlin, Germany, Mar. 25-29, 1996, pp.495-505.
[18] Rothermel G, Untch R H, Chu C, Harrold M J. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 2001, 27(10): 929-948.
[19] Malloy B A, McGregor J D, Krishnaswamy A, Medikonda M. An extensible program representation for object-oriented software. *SIGPLAN Not.*, 1994, 29(12): 38-47.
[20] Liang D, Harrold M J. Slicing objects using system dependence graphs. In *Proc. the International Conference on Software Maintenance* (*ICSM 1998*), Bethesda, USA, Nov. 16-19, 1998, pp.358-367.
[21] Cheung R C. A user-oriented software reliability model. *IEEE Trans. Software Eng.*, March 1980, 6(2): 118-125.
[22] Mills H D, Dyer M, Linger R C. Cleanroom software engineering. *IEEE Software*, 1987, 4(5): 19-25.
[23] Gittens M. The extended operational profile model for usage-based software testing [Ph.D. Dissertation]. Faculty of Graduate Studies, University of Western Ontario, 2004.
[24] Whittaker J A, Thomason M G. A Markov chain model for statistical software testing. *IEEE Trans. Softw. Eng.*, 1994, 20(10): 812-824.
[25] Jacobson I, Christerson M. Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley, 1992.
[26] Procedures for performing a failure mode, effects, and criticality analysis. Department of Defense, US MIL STD 1629A/Notice 2, Nov. 1984.
[27] Mohapatra D P, Mall R, Kumar R. An overview of slicing techniques for object-oriented programs. *Informatica*, 2006, 30(2): 253-277.
[28] Henry S, Kafura D. Software structure metrics based on information flow. *IEEE Trans. Software Engineering*, 1981, 7(5): 510-518.
[29] Boehm B, Clark B, Horowitz E, Westland C, Madachy R, Selby R. Cost models for future software life cycle processes: Cocomo 2.0. *Annals of Software Engineering*, 1995, 1(1): 57-94.
[30] Point O. http://sunset.usc.edu/csse/research/cocomoii/cocomo main.html, 2008.
[31] John S K, Clark J A, Mcdermid J A. Class mutation: Mutation testing for object-oriented programs. In *Proc. Net. ObjectDays*, 2000, pp.9-12.
[32] Li J J. Prioritize code for testing to improve code coverage of complex software. In *Proc. the 16th IEEE International Symposium on Software Reliability Engineering* (*ISSRE 2005*), Chicago, USA, Nov. 8-11, 2005, pp.75-84.
[33] Li J, Weiss D, Yee H. Code-coverage guided prioritized test generation. *Information and Software Technology*, 2006, 48(12): 1187-1198.
[34] Littlewood B. A reliability model for systems with Markov structure. *Journal of the Royal Statistical Society*, Series C (Applied Statistics), 1975, 24(2): 172-177.
[35] Booth T. Performance optimization of software systems processing information sequences modeled by probabilistic languages. *IEEE Transactions on Software Engineering*, Jan. 1979, 5(1): 31-44.
[36] Lo J H, Kuo S Y, Lyu M R, Huang C Y. Optimal resource allocation and reliability analysis for component-based software applications. In *Proc. the 26th Annual International Computer Software and Applications Conference* (*COMPSAC 2002*), Oxford, England, Aug. 26-29, 2002, pp.7-12.
[37] Elbaum S, Malishevsky A G, Rothermel G. Test case prioritization: A family of empirical studies. *IEEE Trans. Softw. Eng.*, 2002, 28(2): 159-182.
[38] Elbaum S, Malishevsky A, Rothermel G. Incorporating varying test costs and fault severities into test case prioritization. In *Proc. the 23rd International Conference on Software Engineering* (*ICSE 2001*), Toronto, Canada, May 12-19, 2001, pp.329-338.
[39] Jeffrey D, Gupta N. Experiments with test case prioritization using relevant slices. *J. Syst. Softw.*, 2008, 81(2): 196-221.

Mitrabinda Ray received the M.Tech. degree in computer science and engineering from National Institute of Technology, Rourkela, India. Currently, she is a Ph.D. candidate at National Institute of Technology, Rourkela, India. Her research interests are software testing and software risk assessment.

Kanhaiya lal Kumawat is a final year B.Tech. student in Computer Science and Engineering Branch at National Institute of Technology, Rourkela, India.



Durga Prasad Mohapatra received his Ph.D. degree from Indian Institute of Technology and M.E. degree from Regional Engineering College (now NIT), Rourkela. He joined the Faculty of the Department of Computer Science and Engineering at the National Institute of Technology, Rourkela in 1996, where he is now an associate professor. His research interests include software engineering, real-time systems, discrete mathematics and distributed computing. He has published more than thirty papers in these fields. Dr. Mohapatra has been teaching software engineering and discrete mathematics to UG and PG students at NIT Rourkela for the past ten years. He has received Young Scientist Award for the year 2006 by Orissa Bigyan Academy. Currently, he is a member of IEEE. Dr. Mohapatra has coauthored the book Elements of Discrete Mathematics: A Computer Oriented Approach published by Tata Mc-Graw Hill.