Journal of Theoretical and Applied Information Technology
31st August 2015. Vol.78. No.3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

TEST CASE GENERATION AND PRIORITIZATION BASED ON UML BEHAVIORAL MODELS

[1]AJAY KUMAR JENA, [2]SANTOSH KUMAR SWAIN, [3]DURGA PRASAD MOHAPATRA

[1]School of Computer Engineering, KIIT University, Bhubaneswar, India

[2]School of Computer Engineering, KIIT University, Bhubaneswar, India

[3]Department of Computer Engineering, National Institute of Technology, Rourkela, India

E-mail: [1]ajay.bbs.in@gmail.com, [2]sswainfcs@kiit.ac.in, [3]durga@nitrkl.ac.in

ABSTRACT

Test case prioritization (TCP) techniques have been proven to be beneficial for improving testing activities. Prioritized test suites are found using different techniques of prioritization. While code coverage based prioritization techniques are found to be used by most scholars, test case prioritization based on UML behavioral models has not been given much attention so far. We propose a novel approach for generating and prioritizing test cases using UML sequence and interaction overview diagrams. First, we convert the interaction overview diagram to interaction Graph (IG) and sequence diagram to Message Sequence Dependency Graph (MSDG). An intermediate graph known as Sequence Interaction Graph (SIG) is generated by combining MSDG and IG. From SIG, we generate the test scenarios and subsequently the test cases. For test case prioritization, the first task is to convert the SIG into another graph known as Dependency Graph (DG). Then, we assign weights to each node of the DG according to its impact using backward slices and based on condition criticality. Further, we calculate the weights of each basic path. According to these weights, finally, we prioritize the test cases. We have used the impact, complexity and criticality of the elements present in the model for the prioritization of test cases. The results obtained by our approach indicates that the proposed technique is efficient and effective in prioritizing (ordering) the test cases using UML models. Our proposed approach achieves higher Average Percentage of Fault Detection (APFD) compared to other related approaches.

Keywords: *Test case, Backward Slicing, Dependency Graph, Code Coverage, Prioritization, Interaction Graph.*

1. INTRODUCTION

Testing of software is the most practical means of ensuring quality of the software in real life applications [21]. It accounts a lion's share in the effort of software development for the current need sophisticated software [1]. Software quality is primarily carried out by means of testing which faces major constraints of effort, time and resources. There is a pressing need for scheduling of test cases effectively within the constraints in order to maximize the throughput i.e. rate of fault detection and coverage of testing. It accounts almost 50% share in the development cycle and has a great impact on the reliability and quality [1]. But, in practical it is very difficult to have exhaustive testing for large software product. So, there is always a challenge to device a new methodology for testing the increased size and complexity of software in the software development lifecycle. Further, as per the need of the society, sophisticated systems are developed every now and then which encourage the software industries to design better quality and reliable software in a fixed duration. So, developers have minimum time to ensure the proper quality of the developed software. A number of challenges have occurred unexpectedly, to meet the need of faster development. Out of these, first important challenge is test case generation at the beginning stage of the development process, so that coding and testing can run parallel. Secondly, focusing on those parts of the coding, which are of higher impact or criticality in view of its priority so that quality software product can be developed with a relatively small testing effort and comparatively in less time. Using the design models of UML, several research attempts have been reported for the generation of test cases [2, 3, 16] for addressing the

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

present challenges. On the contrary, to meet the second challenge it is necessary to organise the test cases in a specific fashion or prioritize. Prioritization of test cases schedules the test cases in such a way that achieve some goals at faster rate.

The increasing size and complexity of recent software products make extensive testing very difficult. Looking to the requirements, the cost and time of testing the software must be planned in an optimized manner and also well in advance. Therefore, it is a challenge to generate test cases from this bulky and complex software and employ them to achieve maximum throughput by uncovering the flaws. For ensuring the quality of the software, test case prioritization is a well known and efficient technique. Among the objectives, the rate of fault detection is the primary one in test case prioritization. By prioritizing the test cases an early detection of flaws is possible in the testing process. Other objectives of test case prioritization include increasing the test coverage in the system under test (SUT) at a faster rate and to increase the confidence in the reliability of the system at a faster rate.

Different methods of test case prioritization are available in literature, e.g. code-based test prioritization and model-based test prioritization reported in literature [11–15]. These techniques consider the knowledge of previous usage of the system such as fault proneness of the different pieces of program code, fault detection capability/fault exposing potential of each test case and certain coverage criteria such as statement, function, relevant slice, paths, data flow and control flow information etc. Most of the test case prioritization methods [9, 14, 19, 20, 23] are code-based and are mainly used for regression testing. These methods are suitable only at the post implementation phase of the software development process. In the code-based test prioritization techniques, the source code of the system is used to prioritize the tests. In model-based test prioritization techniques [17, 18], the system's model is used to prioritize the tests. System models are not only used to design the system under development but also to test and prioritize the tests [17, 18]. In fact, test case prioritization using design models is scarcely reported in the literature [18, 19]. Model-Based Testing (MBT) is more appropriate for object-oriented software than conventional testing because of its high potential for automation, ease of accommodating changes and maintenance [26, 27]. Models are the visual artifacts to analyze complex systems, mainly used for capturing the information about the software system, and the advantage of being reused in the development progresses.

The process of model based test case generation is basically a traversal of the models. The following benefits are obtained by generating test cases and doing prioritization of those test cases from design model:

I. Prior generation of test scenarios leads to clear understanding of the requirements of the user.
II. Prior prioritization of scenarios leads to more concentration on planning, designing, coding, testing and maintaining the product more efficiently.

UML has now become the de facto standard for object-oriented modeling and design [19]. UML models are the important source of information for test case design, which if satisfactorily exploited, can go a long way in reducing testing cost and effort and also at the same in time improving the software quality [9]. UML-based automatic test generation and prioritization are practically important and theoretically challenging topic and are receiving increasing attention from researchers. Traditionally, a large amount of effort was spent to generate test cases from UML diagrams using heuristic based techniques [21]. Very less attempts are taken to prioritize the test cases based on UML diagrams [18, 19].

The behavior of an use case can be represented by using the UML interaction, activity and state chart diagrams. The UML diagrams like sequence shows the sequence of message interactions among different objects as well as interaction among users and objects involved in the software under test (SUT). Sequence diagrams capture the exchange of messages between objects during its execution. It focuses on the order in which the messages are sent. The response of the system is shown through messages between the objects within the system. A path sequence in a sequence diagram represents the set of messages from the start message to end message related to a scenario of an use case. Sequence diagrams can be used to explore the logic of a complex operation, function or procedure. One way to think of sequence diagrams is particularly highly detailed diagrams, is as visual object code. Sequence diagrams describe interactions among software components, and thus are considered to be a good source for cluster level testing. In the rest of the paper, we use the terms message and method interchangeably. We assume that each sequence

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

diagram represents a complete trace of messages during the execution of a user-level operation, which invoke the methods associated with the class of any object within the scenario of the use case. On the other hand, interaction overview diagrams focus on the flow of control among objects. These are very helpful for visualizing the manner how several objects collaborate and control flow of the activities to perform a job. These are also helpful for describing the procedural flow of control and parameter through several activities. The purpose of prioritizing the scenarios observed from sequence and interaction overview diagram is to identify the relative importance of message-method-interaction scenarios. While message is related to the method of any object within the scenario, method is represented by group of activities arranged in a control flow structure to perform the action of the message. The sequence diagram represents high level design to integrate the object code, while the activity diagram is closely related to the inside structure of code. So detail and comparatively accurate weight can be calculated by adding the path of the graph representing sequence diagram with interaction overview diagrams. For test case generation and prioritization, we have gone down to one level for uncovering more faults and efficient prioritization by merging activities in interaction overview diagram with sequence of messages of sequence diagram.

Prioritization of the test cases means ordering or scheduling of test cases based on certain coverage criteria [4, 6, 9, 15]. We have used UML sequence and interaction overview diagrams for the above purposes. First, we convert interaction overview diagram to interaction Graph (IG) and sequence diagram to Message Sequence Dependency Graph(MSDG). An intermediate graph is generated known as Sequence Interaction Graph (SIG) by combining MSDG and IG. From SIG different scenarios are generated and it follows the test cases. Once again, we transform the SIG into an intermediate graph, called Dependency Graph (DG). We use the DG to prioritize testing paths. For prioritization, we calculated the weights of different test cases based on a fitness function which calculates the nodes and paths covered by the test cases. We have used the impact, complexity and criticality of the elements present in the model for the prioritization of test cases. The results obtained by our approach indicated that the technique is effective and efficient in ordering of test cases using combined models of UML. We have used backward slice of each method or activity to calculate the impact (influence) of a method or activity in a use case scenario. A method may influence one or more methods and other statements of the program. If the impact (influence) of the method is more, then method is more critical. If more number of elements uses the output of a method or activity, then an error in the output will affect more. Therefore, special care should be taken for those elements during testing for early detection of error and reliability of the software. These elements should not only be tested more thoroughly composed to other elements at the time of unit level but also at integration (cluster) level because the interface faults can not be detected during unit testing. We have designed a prioritization metric using impact (influence) of a method and criticality of conditions for executing a method/activity within a scenario of object-oriented software. The determination of criticality and complexity of conditions (edges) also helps in reliability measure of the software. The proposed metric is also a measure of criticality and severity of the test cases.

Program slicing helps in understanding programs by dividing it into slices, so the task of testing can be allocated to various testers. Each tester can test the slice in the program domain. After introduction of slicing concept by Wiser [33], researchers have shown special interest in this promising research area for its wide application. Slicing has been found to be useful in several important application areas such as software testing, maintenance, reengineering, decomposition and integration, and debugging. Program slicing is essentially a decomposition technique that extracts only those program statements that are relevant to a particular computation [33, 34]. A program slice is constructed with respect to a slicing criterion. A slicing criterion (l, V) specifies a location (l) of a statement and a set of variables (V) in that statement. A slice of a program P with respect to a slicing criterion (l, V) is the set of all statements of the program P that might affect the slicing criterion for every possible input to the program. There are two types of slicing techniques, static slicing, dynamic slicing depending upon the run time environment where as depending upon the graphical traversal environment it is divided as forward slicing and backward slicing. Here we used backward slicing for test case prioritization.

The rest of the paper is organized as follows. Some basic concepts are presented in Section 2. The review of some existing work in test case prioritization is presented in Section 3. We

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

presented our approach to generate the test cases in Section 4, subsequently prioritization of test cases. An analysis of the approach with experimental results are presented in Section 5 with a case study to illustrate the approach. Experimental results are also presented using Average Percentage of Fault Detected (APFD) in this section. Section 6 describes the comparison of our proposed approach with other existing related work. Section 7 presents the future work and conclusion.

2. BASIC CONCEPTS

In this Section, we discuss some useful definitions which are used in our approach. We also discuss the basic concepts and technique of slicing used in our approach.

Rothermel et al. [4, 6, 8] had given a formal definition for test case prioritization:
Given: TS, is a test suite; PR, the set of permutations of the test suite TS; f is a function from the permutations PR to the real numbers R.

$$f : PR \rightarrow R$$

Problem: Find $TS' \in PR$ such that $(\forall\ TS'')\ (TS'' \in PR)\ (TS'' \neq TS')\ [f(TS') \geq f(TS'')]$.

where, PR is the set of all possible orders of TSs, and f is an objective function that is applied to any such order.

Test Flow Graph: A Test Flow Graph (TFG) G of a diagram D is a flow graph denoted as a quadruple (N, E, S, F) where each node $v \in N$ represents either a message or activity and an edge $e \in E$ represents a transition between the corresponding nodes. An edge $(m, n) \in E$ indicates the possible flow of control from node m to the node n. Node S represents the entry node and F is the set of exit nodes of the diagram D. A Test Flow Graph (TFG) generated from sequence diagram represents the possible message/method sequences in an interaction.
Path: A *path P* from the *start node* to an *end node* in TFG is a sequence of nodes and edges in the TFG.

Message Path Coverage Criterion: A message path in a *sequence diagram* (SD) is a set of sequence messages that begins with an outwardly generated event and terminates with the production of a reply that satisfies this event. A requirement of adequate testing based on *sequence diagram*s is that all the start to end message paths in the diagram are covered by test executions.

Interaction Path Coverage: Given a test set T and Interaction overview *Diagram* IOD, T must cause each possible interaction path in IOD to be taken at least once. An interaction Path is any sequence of activities from the initial interaction to the terminal interaction in the interaction overview *diagram*.

Message-activity path coverage: Let *P* be a set of *paths* obtained from TFG of sequence and interaction overview diagram. Let *TC* be a set of test cases. TC must cause each possible message path in sequence diagram SD with corresponding interaction path in interaction overview diagram IOD to be taken at least once. For each *P* there must be one test case $tc \in TC$ such that when the use case is executed using *tc*, *path P* of TFG is exercised.

Dependency Graph (DG): It is a directed graph. DG= {N, D} where N is the set of nodes and D is the set of directed edges. Here N= {M, A}, where M is the set of message-method nodes and I is the set of interactions of the corresponding interaction overview diagram. D denotes the edge with guard condition. Edges of DG called as dependency edges represent dependencies among nodes. Two types of edges are present in DG: control dependence and parameter dependence edges. In parameter dependence, the edge between two nodes implies that the computation performed at the node pointed by the edge which is directly depend on the value computed at the other node. A control dependence edge between two nodes implies that the result of the predicate expression at the node pointed by the edge decides whether to execute the other node or not.
Dependency Edge: If a node $i \in N$ is either data or control dependent on node $j \in N$, then there is an edge in DG from node j to node i called a dependency edge.
Slicing: It is essential to find the statements that might be affected by a variable at some point in the program [35]. This can be obtained by proceeding forward over the Program Dependency Graph (PDG) to find all the nodes that have directly or indirectly affected the value of the variable.
Backward Slicing: It is essential to find the statements that might be affected by a value of a variable at some point in the program [35]. This can be obtained by proceeding backward in the Program Dependency Graph (PDG) to find all nodes that have directly or indirectly affected by the value of the variable at some point of interest

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

[35]. The backward slice contains the statements of the program, which can have some effect on the slicing criterion, where as a forward slice contains those statements of a program, which are affected by the slicing criterion. Backward slices can assist a developer by helping to locate the parts of a program that will be affected by a modification.

Ex.
Let us consider a small fragment for forward and backward slicing :

```
S1 :    VAR
S2 :    a,b,c: INTEGER;
S3 :    BEGIN
S4 :    a := 1;
S5 :    b := a + 2;
S6 :    c := b + 3;
S7 :    END
```

Output of Forward slice w.r.t. (4, a) is

```
b : = a + 2;
c : = b + 3;
```

Output of Backward slice w.r.t. (6, c) is

```
a  : = 1;
b : = a + 2;
```

3. RELATED WORK

In this section we, present different test case generation and prioritization techniques using the models of UML. Then, a discussion of the existing test case prioritization techniques is described.

To accomplish specific goals, prioritization is the arrangement of the test cases according to some coverage criteria is reported in some literatures [4, 6, 7, 9, 10-14]. Test case prioritization mainly done in two ways i.e. based on the source code or from requirement specifications. The work reported in [11, 12] use requirements as basis for test case prioritization. The authors assigned weights to the factors such as Customer-assigned Priority on Requirements to prioritize the test cases.

Some other methods for test case prioritization [4, 7] are based on the source code, which look at structural and data coverage. They compare prioritization techniques, based on granularity, namely, *fine-grained* and *coarse-grained*. They used the weighted Average Percentage of Faults Detected (APFD) metric for measuring effectiveness. Elbaum et al. [6] tried to improve the rate of fault detection for their prioritization technique. By using the statement and function level coverage they prioritize the test cases. In statement coverage technique, they considered prioritization of test cases based on the coverage of maximum statements, coverage of statements not yet covered, probability of exposing faults, etc. Functional coverage-based criterion is analogous to statement based coverage criterion, except that it operates at the level of functions. Elbaum et al. also consider a metric called average of the percentage of faults detected (APFD) to measure the effectiveness of these prioritization techniques. Their experimental result showed that the statement-based prioritization techniques were more effective than the function-based prioritization techniques. APFD metric gives better result under two assumptions: all faults were of equal severity and all test cases have equal costs. To cope with this limitation, Elbaum et al. [10] proposed a new cost-cognizant metric APFDC. This new metric considered both the percentage of total test case cost generated out of the process and detection of the percentage of total fault severity.

A new approach for prioritization of the test cases in regression testing was reported by Jeffrey and Gupta [9]. They considered the statements that were affected or had possibility to affect the output of the program by using the relevant slice of the output. Further they proposed that if any modification performed in the program has an outcome on the output of a test case in the regression test suite i.e it must affect some computations in the relevant slice of the output. Subsequently they proposed a heuristic, that assigned weight to a test case with larger number of statements (branches) in its relevant slice of the output.

By using the state-based approach, Korel et al. [17] proposed a test case prioritization technique for regression testing. They proposed two prioritization techniques i.e selective test prioritization and model dependence-based test prioritization to find the known faults of a modified system. For achieving higher rate of fault detection, Korel et al. [17] used both the models and found a difference between them. After executing the whole test suite against the modified model they collected information related to the difference. From their observations, they noticed that model dependence-based test case prioritization significantly performs the selective test prioritization so far as the early fault detection rate is concerned. But in comparison to selective test case prioritization, model dependence-based test case prioritization is very complex and requires huge amount of storage space and also difficult to

Journal of Theoretical and Applied Information Technology
31st August 2015. Vol.78. No.3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

implement in critical systems. In their next work, Korel et al. [17] extended the earlier work by introducing five new heuristics to reduce the high storage requirement.

Kundu et al. [23] proposed three prioritization metrics by using the model information in the sequence diagram. Their approach satisfied the scenario coverage criterion and were suitable for system-level testing. The values of these prioritization metrics can be analytically computed from the model information only. In their work, they presented an approach to generate test data using rule-based matrix. The prioritization metrics are used to control the number of test data without compromising the test adequacy.

Panigrahi et al. [31] proposed an intermediate graph of the source code of the program for the prioritization of test cases. After a partial change in a program the model is updated. Accordingly the nodes in the graph are affected. They used slicing for the affected nodes for test case prioritization

Sapna et al. [32] proposed a technique to prioritize the test cases using the activity diagram. They converted the activity diagram to form a tree structure. By using depth first search traversal, they found the scenarios. After that the authors assigned higher weights to the fork-join nodes, then lesser weights were assigned to branch-merge nodes and lowest weights were assigned to action/activity nodes. By arranging the path weights in decreasing manner, they prioritized the test cases. By using a case study, they demonstrated the working of their technique.

The test case prioritization technique suggested by P.R. Srivastava [36] helps in finding out the average faults found out per minute. He proposed an algorithm to determine the effectiveness of the prioritized and non-prioritized test cases by calculating the Average Percentage of Faults Detected (APFD) in regression testing. Mahali et al. [29] proposed a genetic algorithmic approach to prioritize the test cases. They used a case study of a Shopping mall and drawn the activity diagram and subsequently found the critical path from the graph constructed form the activity diagram. They found the optimized critical path having maximum value which leads to prioritize the test cases.

4. PROPOSED APPROACH

Each use case of a system can be represented with one or more sequence and interaction overview diagrams. In our proposed approach, we employ sequence and interaction overview diagrams to represent the requirements of a system to develop the scenarios. Each scenario is a complete path through the Sequence Interaction Graph (SIG), which is generated from interaction overview and sequence diagrams of the system under consideration. Users of the system can traverse many paths for generating test cases. The main scenario begins from the start node; by traversing through all intermediate nodes without any fault, up to the end node. Alternate scenarios from alternate paths are the cases when there is wrong entry of input or a condition is not satisfied. This research work proposes an approach for prioritizing test cases meant for testing of software using object-oriented approach.

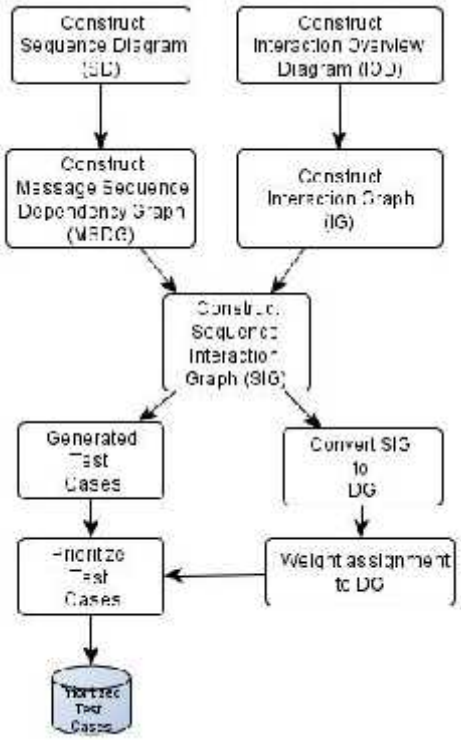


Figure 1: Proposed methodology for test case prioritization (TCPN)

We propose a framework named as Test Case Prioritization (TCPN) which is shown in Figure 1. The input to TCPN is an interaction overview diagram and a sequence diagram depicting one scenario of a use case. As given in Figure 1, TCPN

Journal of Theoretical and Applied Information Technology
31st August 2015. Vol.78. No.3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

consists of three tasks. For test case generation, we first convert interaction overview and sequence diagram into Interaction Graph (IG) and Message Sequence Dependency Graph (MSDG). Then, by combining IG and MSDG we prepared an intermediate graph called Sequence Interaction Graph (SIG). We traverse the SIG using depth first search (DFS) algorithm for basic path generation. Next, the test cases are generated corresponding to each basic path. For test case prioritization, the first task converts the SIG into Dependency Graph (DG). Then we assign weights to each node of DG according to its impact using backward slices and a fitness function. Then, we calculate the weights of each paths. Finally, we prioritize the generated test cases according to weights of the paths. If multiple test cases are having the same weights, we can order them according to the severity/risk of requirement of test cases [33].

4.1 Case Study

We have considered Library Book Issue as a case study to explain the working of our proposed approach. The Sequence Diagram and Interaction Overview Diagram of Library Book Issue System are created using StarUML and presented in Figure 2 and Figure 4 respectively. The users coming to the library for issuing books are considered in our use case. The user may or may not be a valid user of the library. The books in the library may or not be available. The books can be issued to the user if he/she is a valid member, books are available and he/she has not issued the number of books he/she is entitled for. Accordingly the error messages will be displayed. If the books will be issued to the user, the book status and member records will be updated. The transaction will also be recorded.

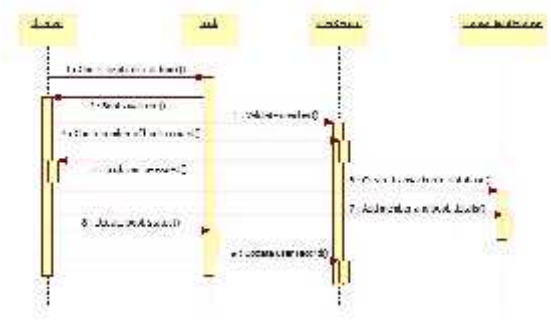


Figure 2: Sequence Diagram of Library book issue use case of LIS

Figure 3: Snapshot of XMI code of SD

By using Table 1 and the XMI code in Figure 3, we generate MSDG of the sequence diagram as presented in Figure 5. Then, we try to get the ID of the objects from the XMI code and the sequence of the objects. Similarly, by using Table 2 and the XMI code of IOD, the control flow graph is prepared as given in Figure 6.

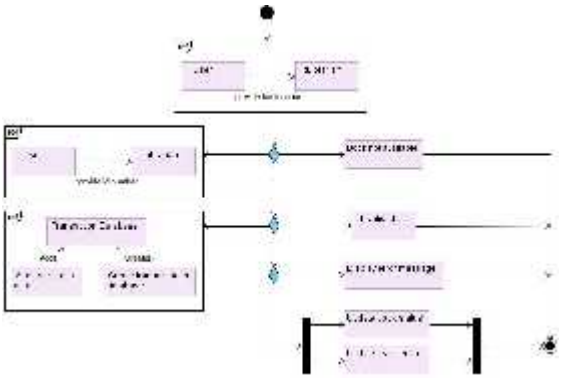


Figure 4: Interaction Overview Diagram (IOD) of Library Book Issue System

Table 1: Message dependency from Sequence Diagram

Symbols	Messages Passed between Objects
S1	Check book availability()
S2	Book available()
S3	Validate member()
S4	Check no of books issued()
S5	Book issued()
S6	Create Transactions()
S7	Add member and Book details()
S8	Update Book Status()
S9	Update User Record()
S10	Disp_Err_Mess1("Book not available")

Journal of Theoretical and Applied Information Technology
31st August 2015. Vol.78. No.3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

S11	Disp_Err_Mess2("Not a valid member")
S12	Disp_Err_Mess3("Books entitled completed")
S13	End

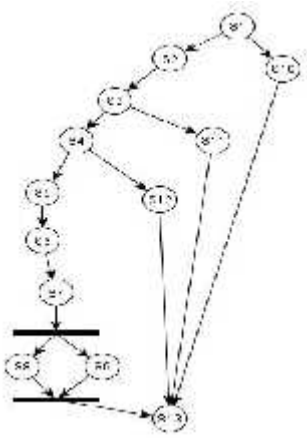


Figure 5: Message Sequence Dependency Graph (MSDG) of sequence diagram

Table 2: Nodes of the IOD with Id_Number

Identification Number	Interaction nodes
I1	Check Book
I2	Book not available
I3	Insert user No
I4	Invalid user
I5	Check no of Books
I6	Display error message
I7	Update book status
I8	Update user record
I9	Issue Book
I10	Stop

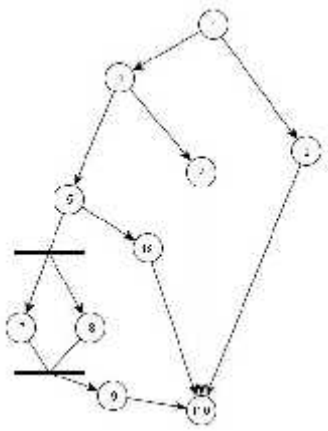


Figure 6: CFG of the IOD given in Figure 2

4.2 Intermediate Representation

For generating the test scenarios, it is necessary to transform the diagrams into suitable intermediate representations. The intermediate Sequence Interaction Graph (SIG) generated by using the message dependency graph given in Figure 6 and the control flow graph given in Figure 7, is presented in Figure 8(a). SIG is generated from the sequence diagram and represents the possible message/methods between sequences in an interaction. A SIG is a graph G = (V,E), where V is the set of nodes, and E is the set of edges. Nodes of SIG represent the messages and edges represent the transitions.

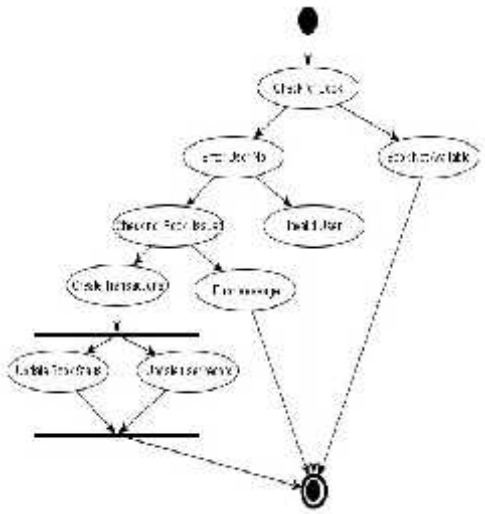


Figure 7: SIG of Library book issue use case

Journal of Theoretical and Applied Information Technology
31st August 2015. Vol.78. No.3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

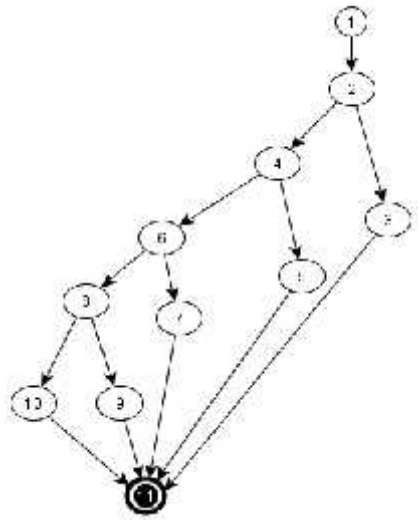


Figure 8: SIG of Library book issue use case

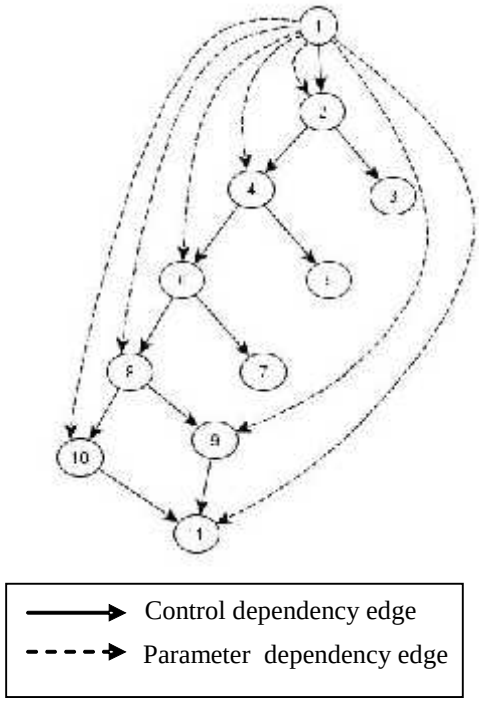


Figure 9: Dependency Graph (DG) of Figure 8

The message that initiates the interaction is considered as the root of the graph. In an IOD, each transition is labelled with a guard condition. The conditional predicate corresponding to the guard condition might trivially be an empty predicate which is always true. The initial nodes in the sequence and IOD diagrams are same. For the intermediate representation, all the interactions from the IOD are considered first from Figure 7 and all sequences of interaction are considered from MSDG of Figure 6. Figure 9 shows the DG of the Sequence Interaction Graph. The solid directed edges represent the control dependency and the doted edges represent the parameter dependency edge.

4.3 Test Case Generation

For the generation of test cases, we have merged activities in the interaction overview diagram with sequence of messages in the sequence diagram. For this, we merged IG of interaction overview diagram with MSDG of sequence diagram into a single SIG. The first node of IG of corresponding activity diagram is merged with the called message node. Then last node of IG of activity diagram is merged with the arrow-head pointed node of the corresponding called node in the MSDG of sequence diagram. The numbers associated with each IG and MSDG node are called node identifiers. The node identifier indicates the message number or interaction code of the sequence or interaction overview diagram respectively. Figure 8 shows the merged diagram or SIG generated from sequence and interaction overview diagrams. From SIG by adding parameter dependency edges, we constructed dependency graph (DG) which is shown in Figure 9.

To generate test cases that satisfy the message-interaction path criteria, we consider all the possible paths from the start node to a final node of the SIG. Then, each path is visited to generate test cases. During the visit, we look for conditional predicates on each of the transitions. For each conditional predicate, we generate test cases using category-partitioning method [28].

With the testing scenario, all possible sequences of object message interactions and activity paths (one possible sequence of message, activity and edge corresponds to one path) are covered to verify whether sequence of message-interaction paths is correct or not. By covering all possible sequences of message interactions with the activity path coverage criterion, it is ensured that all user inputs, all object responses and all possible activity path sequences (that can be covered solely by the branch, condition and statement coverage criterion) are covered. Therefore, the generated test cases confirm the adequacy of message-interaction path

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

coverage.

From the SIG of Library book issue use case, we generate five paths as given in Figure 8. The independent paths are

Path1 – $1 \rightarrow 2 \rightarrow 3 \rightarrow 11$
Path2 – $1 \rightarrow 2 \rightarrow 4 \rightarrow 5 \rightarrow 11$
Path3 – $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 7 \rightarrow 11$
Path4 – $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 8 \rightarrow 10 \rightarrow 11$
Path5 – $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 8 \rightarrow 9 \rightarrow 11$

The generated test cases using these independent paths are shown in Table 3.

4.4 Test Case Prioritization

The In this section we, discuss our proposed approach to prioritize the test cases generated from UML sequence and interaction overview diagrams.

Test case prioritization achieves many possible goals. Out of these goals, we restrict our attention to devise techniques that will improve the efficiency by early detection of faults and provide confidence in reliability with good testing coverage. To formally illustrate how rapidly a prioritized test suite detects faults, Rothermel et al. [24] introduced a metric called Average Percentage of Faults Detected (APFD) to measure the weighted average of the percentage of faults detected during the execution of the test suite [4]. Let T be a test suite which contains n test cases, and let F be a set of m faults revealed by T. Let TF_i be the first test case in ordering T′ of T which reveals fault i.

According to [24], the APFD for test suite T′ is as follows :

$$APFD = 1 - \frac{TF_1 + TF_2 + ... + TF_m}{nm} + \frac{1}{2n} \qquad (1)$$

where m is the number of faults contained in the program under test P and n is the total number of test cases.

Always the value of APFD will range from 0-100. The higher is the APFD value, the faster (better) will be the fault detection capability in the test suite.

Let us consider a path P_i in a SIG. Let the test case T_i corresponds to path P_i. The prioritization technique assigns weights to each of the path based on the backward slice and criticality of nodes. We propose a metric to guide the test case prioritization using message/interaction of nodes. Backward slice of a path is used to calculate the impact of interaction upon other message/interactions. Accordingly the weight of the path P_i computed. After knowing the values for all paths in SIG, then these values will be assigned to the associated test cases. The prioritized test cases will be obtained by arranging the test cases in the decreasing order of weight.

The impact of a path of SIG will be calculated from the number of nodes affected by the current node. The number of nodes affected by a node is calculated using backward slice of a node. The algorithm for computing backward static slices of all method and interaction nodes is given in Figure 6. We have named this as *Calc_Backward_Affect* algorithm. An empty set is assigned to each node of SIG before the algorithm is applied. After that, the set associated with the node N_i will contain all method/interaction nodes, which are affected by node N_i. Since the function *Slice_Back( )* is invoked recursively the algorithm builds the set of nodes associated with each node in SIG incrementally. The function *Slice_Back( )* is executed with the node value N_i which is passed as an argument. If any unvisited node is found then it is marked as visited, the node identifier is added to the set associated with node N_i ,and all outgoing edges form node N_i are traversed forward. If an outgoing edge is attached to a visited node N_i, the node identifiers included in the set associated with node N_i are added to the set associated with node N_i. Otherwise, if the outgoing edge is attached to a node N_j not yet visited, node N_j is passed as an argument to the *Slice_Back( )*. The function *Slice_Back ( )* finds the set of nodes included in the backward slice computed at node N_i. Then, the node identifiers included in the set associated with node N_j are added to the set associated with node N_i.

The algorithm *Calc_Backward_Affetc* ensures that every edge is traversed once and then it is marked visited. Before visiting each node is marked not visited. At the time when a node N_i is passed to the *Slice_Back( )*, it checks whether the node is marked visited or not then it proceeds. If the marked node is not visited, *Slice_Back* function makes a mark on the node as visited. Then it traverses all the outgoing edges from the node and if the node visited earlier, the *Slice_Back( )* will be terminated. So, the outgoing edges of a node are traversed only once. As a result we have seen no edges are visited more than once, when the

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Calc_Backward_Affect algorithm is applied. The number of nodes included in the backward slice of a node N_i will be considered as *impact* of that node N_i.

Algorithm *Calc_Backward_Affect*
Input: *A SIG with start node P, a slicing criterion (s)*
Output: *Backward slice for every node*

1. $S_i[..] = \Phi$ // S_i *is the set associated with* N_i
2. $MV_i = 0 \ \forall i.$ // V_i *is status of visiting node* N_i
3. *Slice_Back* (N_i)
4. *{*
5. *if* $MV_i = 1$ *then*
6. *{*
7. *exit (0) // to come out after nodes visited*
8. *}*
9. *else*
10. *{*
11. $MV_i = 1$ // N_i *visited*
12. $Y_i = \{N_i | N_j$ *depends on* $N_i \}$
13. $S_i = S_i \cup Y_i$
14. *for (each node* $N_j \in Y_i$*)*
15. *Slice_Back*(N_j) *// recursively called the fun*
16. *}*
17. *endif*
18. *}*

4.4.1 Proposed test case prioritization technique

In this Section, we describe the steps of the proposed prioritization technique.

i. Traversing the paths in the graph SIG using depth first search principle.
ii. Assigning the costs to the nodes using backward slice of the affected nodes.
iii. Assigning the costs to the edges based on 80-20 rule.
iv. Calculating the costs of the paths as per nodes and edges.
v. Arranging the paths to get the prioritize order.

Assigning the costs to the nodes of SIG:

Each node of the graph is assigned with a cost as per the number of nodes affected by the current node using backward slicing given in the algorithm *Calc_Backward_Affect*. By using this algorithm, we have calculated the backward slice of each node of the SIG in Figure 8 and shown in Table 4.

Table 4: Cost of each node using back slices

Node	Backward Slice	Cost
1	1,2,3,4,5,6,7,8,9,10,11	11
2	2,3,4,5,6,7,8,9,10,11	10
3	3,11	2
4	4,5,6,7,8,9,10,11	8
5	5,11	2
6	6,7,8,9,10,11	6
7	7,11	2
8	8,9,10,11	4
9	9,11	2
10	10,11	2
11	11	1

Assigning the costs to the edges of SIG

For assigning the cost of each edge in the graph, we apply 80-20 rule, where 80% of the actions are passed to the truth value part when it comes across a decision node and 20% of the actions are passed to the false value part. Here we considered the total cost as 5 while passing near a decision node. So, cost 4 is assigned to the truth value part (80%) and 1 is passed to the false part (20%). Cost 2 is assigned to the normal edges. After assigning the costs to edges of the SIG, the weighted SIG is given in Figure 10.

$$Cost(edge) = \begin{cases} 4, & \text{for the truth value of the decision node} \\ 1, & \text{for the false value of the decision node} \\ 2, & \text{otherwise, normal edges} \end{cases}$$

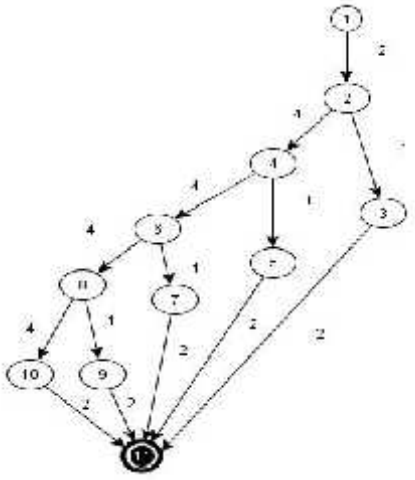


Figure 10: SIG after assigning the cost

Journal of Theoretical and Applied Information Technology
31st August 2015. Vol.78. No.3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

The cost of each path can be calculated by the following equation

To calculate the cost of each path, we use Equation 2. The cost of each path is assigned with the values of all nodes in the path and the edges along the path. The cost of the edges along the path, cost of the nodes in the path and the total cost of the path in the SIG given in Figure 10 are shown in Table 5. In Table 6 we describe the prioritized order of the test cases.

$$Cost\ (path) = \sum_{i=1}^{n} Cost\ (node_i) + \sum_{j=1}^{n} Cost\ (edge_j) - - - (2)$$

Considering the weights calculated in Table 5, we prioritize the test cases of corresponding basic paths in decreasing order of their calculated costs. Hence, the order of execution of the test cases is T4, T5, T3, T2, T1.

By arranging the costs of each path in descending order, the test cases can be prioritized in the order T6, T3, T5, T2, T4, T1. Now by applying Equation 1 for the prioritized test cases we compute the value of APFD. Table 7 shows the faults detected by the test cases. In Table 6 we observe that, the number of test cases n is 5 and the number of faults (m) is 8. Now, by applying these values in Equation 1, we get

$$APFD = 1 - \frac{1+1+1+2+1+2+1+1}{5*8} + \frac{1}{2*5} = 0.825$$

Now, APFD value for the non-prioritized test case (i.e. T1, T2, T3, T4, T5, T6) can be calculated as follows:

$$APFD = 1 - \frac{1+2+4+2+4+2+1+1}{5*8} + \frac{1}{2*5} = 0.675$$

By comparing the APFD values of the prioritized and non-prioritized test cases, it is observed that, the APFD value obtained for prioritized cases is greater than that of the non-prioritized ones. So, by using our approach, it generates about 15% more effective prioritized test cases than the randomized approaches. This is also illustrated in Figure 12 (a) and Figure 12 (b) for prioritized and non-prioritized test suites.

5. EXPERIMENTAL RESULTS

The In this section we, analyze the performance of our proposed approach for test case prioritization. For establishing the efficiency of the approach, a case study of Library Book Issue System is discussed in Section 4. Then we have shown the efficiency of the proposed method by calculating APFD measures [33].

Test case prioritization techniques organize the test cases in an order to achieve better performance like increased rate of fault detection and higher code coverage, to increase their effectiveness. The faults are equally likely to exist at any statement of the code, which constitutes methods, or primitives of associated messages and activities of the methods in object-oriented software. A message is a request that an object makes to another object to perform an operation. The operation executed as a result of receiving a message is called a method. So, methods of associated messages and activities cover the code. If methods/activities with high dependency are exercised first, they would be expected to uncover more faults. Therefore, the test priority would provide an order to run test suites based on backward slicing information of method or interaction. So the test cases covering methods/interactions at a faster rate are likely to identify the faults in the code at the same speed.

We also implement this approach in other four different case studies from the real-life situations. Table 7 shows the percentage of increase in APFD for prioritized and non-prioritized test cases using our approach over the APFD for the non-prioritized test cases using randomized approach. Figure 11 represents the bar graph showing the average percentage of fault detection of different cases studies by using prioritized test cases and non-prioritized test cases. The x-axis of the graph represents the five different case studies executed using our approach and the y-axis is the values of the APFD.

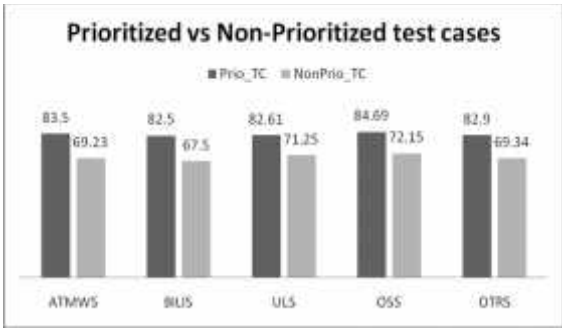


Figure 11: APFD percentage using Prioritized vs non-prioritized test cases

Journal of Theoretical and Applied Information Technology
31st August 2015. Vol.78. No.3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

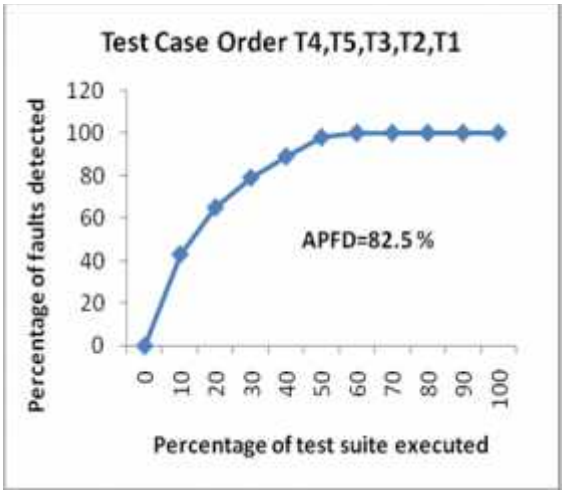
6. COMPARISON WITH RELATED WORK

Most of the test case prioritization techniques [11-15, 31] used for regression testing are based on code. The methods used by the authors [11-15] consider the knowledge of previous usage of the system, such as fault proneness of the different pieces of program code, fault detection capability, fault-exposing potentiality of each test case etc. In this work we proposed an approach by using UML design specification. In code based system the prioritization metric is calculated by using the information of statements, group of statements (branches), slices, functions, data flow and control flow etc. The primary difference in the code based and the proposed model-based approach is that one can apply this technique in early stage of software development prior to its coding and implementation. But, the previous techniques reported in [11-15, 31] can only be applied after implementation. As this approach is applicable before coding, i.e. at design level, so early detection of flaws is possible and better test planning can be done which saves time and cost of the developer.

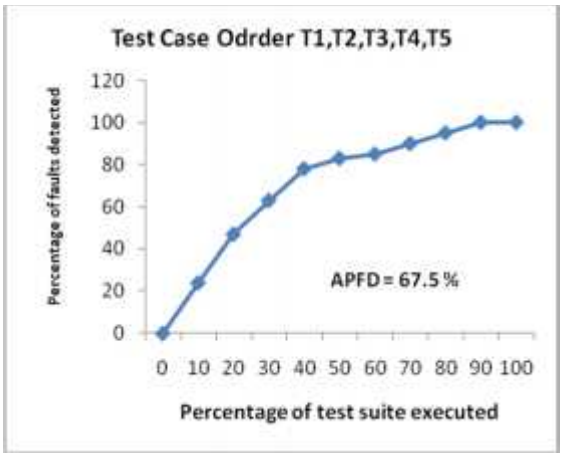
Our work is also comparable with the approach of Korel et al. [18]. In their work [18] the authors tried to capture the model information from a single state-based design specification for the purpose of prioritization. But our work uses multiple designs of sequence and interaction overview diagrams for prioritizing the test cases. Prioritization technique used by the authors [18] is used for regression testing and considers higher rate of detecting the flaws. But our TCP technique, on the contrast is meant for integration testing and achieves the goals like faster rate of code coverage, higher rate of detecting the faults, faster rate of increasing the confidence in the reliability of the system and likelihood of revealing regression errors related to specific code changes earlier in the regression testing process.

In comparison with the approach of Panigrahi et al. [31], our proposed methodology gives more significant result. Panigrahi et al.[31] used a single design model for prioritizing the regression test cases. In case of our approach, we used a combined design model which provides early detection of faults.

We have compared our approach by calculating APFD measure in comparison to the technique proposed by Kundu et al. [19, 23]. Their approach uses one design model where in our approach we use combination of two design models.



(a) Prioritized test cases



(b) Non-prioritize test cases

Figure 12: Example illustrating the APFD measure of prioritized and non-prioritized test cases

7. CONCLUSION

In this paper, we presented a model called TCPN for generating and prioritizing test cases from UML sequence and interaction overview diagrams. We have converted these diagrams into a graph called Sequence Interaction Graph. In literature we found most of the test case prioritization approaches are based on code and are mainly used for regression testing. We proposed a completely model-based approach which is suitable for cluster/integration level testing. The test case generation approach in our work from a sequence and interaction overview diagram is simple and completely systematic and logical. We have proposed the prioritization metric considering the impact or influence of methods and

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

activity and criticality of guard conditions to perform those methods and activities. This prioritization metric is easy to compute. With the proposed approach, some performance goals are achieved which includes faster rate of coverage of code, higher rate of detecting faults, and faster rate in increasing the confidence in reliability of the system. The results obtained from our approach are compared with the approaches of some other researches and observed that by using combined models of UML is more efficient and schedules the test cases in such a order that it detects the faults prior to execution. In future work we would like to optimize the test cases using some soft computing techniques like genetic algorithm, ant colony optimization, particle swarm optimization etc.

REFRENCES:

[1] R. Mall, "Fundamentals of Software Engineering", Prentice-Hall, Springer-Verlag GmbH, 3rd Edition 2009.

[2] A. Bertolino, and F. Basanieri, "A practical approach to UML-based derivation of integration tests", *Proceedings of the 4th International Software Quality Week Europe and International Internet Quality Week Europe (QWE)*, Brussels, Belgium, 2000.

[3] J. Hartmann, C. Imoberdorf, and M. Meisinger, "UML-based integration testing", *In ACM SIGSOFT Software Engineering Notes, Proceedings of International Symposium on Software testing and analysis*, 2000.

[4] S. Elbaum, A.G. Malishevsky, G. Rothermel, "Test case prioritization: A family of empirical studies", *IEEE Transactions on Software Engineering*, Vol. 28, No. 2, February, 2002, pp. 159-182.

[5] C. R. Panigrahi, and R. Mall, "Model-based regression test case prioritization", *ACM SIGSOFT Software Engineering Notes*, Vol. 35, No. 6, November 2010, pp. 1-7.

[6] S. Elbaum, A.G. Malishevsky, and G. Rothermel, "Prioritizing test cases for regression testing", *ACM SIGSOFT Software Engineering Notes*, Vol. 25, No. 5, 2000, pp. 102–112.

[7] P. Tonella, P. Avesani, and A. Susi, "Using the Case based ranking methodology for test case prioritization", *Proceedings of the International Conference on Software Maintenance (ICSM)*, IEEE Computer Society, September 2006, pp. 123-133.

[8] G. Rothermel, R. H. Untch, C. Chu, and M.J. Harrold, "Prioritizing test cases for regression testing", Software Engineering, Vol. 27, No. 10, 2001, pp. 929-948.

[9] D. Jeffrey, and N. Gupta, "Test case prioritization using relevant slices", *International Computer Software and Applications Conference (COMPSAC)*, IEEE Computer Society, Washington, DC, 2006, pp. 411–420.

[10] J.A. Jones, and M. J. Harrold, "Test-suite reduction and prioritization for modified condition/decision coverage", *IEEE International Conference on Software Maintenance (ICSM*), IEEE Computer Society, Washington, DC, 2001, pp. 92–101.

[11] S. Elbaum, A.G. Malishevsky, and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization", *Proceedings of International Conference on Software Engineering (ICSE)*, IEEE Computer Society, Washington, DC, pp. 329–338, 2001.

[12] H. Srikanth, "Requirements-based test case prioritization", *12th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, 2004.

[13] H. Srikanth,and L. Williams, "On the economics of requirements-based test case prioritization", *Proceedings of the 7th International Workshop on Economics-Driven Software Engineering Research*, 2005.

[14] J. Kim, and A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments", *Proceedings of. International Conference on Software Engineering (ICSE)*, ACM, New York, pp. 119-129, 2002.

[15] J.J. Li, D. Weiss, and H. Yee, "Code-coverage guided prioritized test generation", *Information and Software Technology*, Vol. 48, No. 12, 2006, pp. 1187-1198.

[16] J. Offutt, and A. Abdurazik, "Generating tests from UML specifications", *Proceedings of 2nd International Conference on the UML* , 1999, pp. 416-429.

[17] B. Korel, L.H. Tahat, and M. Harman, "Test prioritization using system models", *IEEE International Conference on Software Maintenance (ICSM,*. IEEE Computer Society, Washington, DC, 2005, pp. 559-568.

[18] Korel B., Koutsogiannakis G., Tahat L.H., "Model-based test prioritization heuristic methods and their evaluation", International Workshop on Advances in Model-based Testing, ACM, New York, 2007.

[19] D. Kundu, and D. Samanta, "A novel approach of prioritizing use case scenarios", *Asia–Pacific Software Engineering Conference (APSEC)*, IEEE Computer Society, Washington, DC, 2007, pp. 542-549.

[20] R. V. Binder, "Testing Object-oriented Systems Models, Patterns, and Tools", *Addison-Wesley: Reading*, MA, 1999, pp. 34-43.

[21] S. K. Swain, and R. Mall, "Test Case Generation using UML Sequence and Activity Diagrams", *International Journal of Computing Science and*

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Communication Technologies(IJCSCT), Vol. 1, No. 2, January 2009, pp. 91-100.

[22] J.J. Li, "Prioritize code for testing to improve code coverage of complex software", *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, IEEE Computer Society, Washington, DC, 2005, pp. 75-84.

[23] D. Kundu, M. Sarma, D. Samanta, and R. Mall, "System testing for object-oriented systems with test case prioritization", Software Testing, Verification and Reliability, Published online in Wiley Inter Science (www.interscience.wiley.com). DOI: 10.1002/STVR, 2009.

[24] S. Elbaum, G. Rothermel, S. Kanduri, A.G. Malishevsky, "Selecting a cost-effective test case prioritization technique", Software Quality Journal, Vol. 12, No. 3, September 2004, pp.185-210.

[25] A. Pretschner, "Model-Based Testing in Practice", In Formal Methods (FM'05), Springer-Verlag, Vol. 3582, 2005, pp. 537-541.

[26] I.K. El-Far, J.A. Whittaker, "Model-Based Software Testing", In Encyclopaedia of Software Engineering, 2nd ed, J. J. Marciniak, Ed.: John Wiley & Sons, Inc., 2002.

[27] D.P. Mohapatra, R. Mall, and R. Kumar, "An Overview of Slicing Techniques for Object-Oriented Programs", Informatic , Vol. 30, 2006, pp 253–277.

[28] G. Canfora, A. Cimitile, and A. De Lucia, "Conditioned Program Slicing", Information and Software Technology, Vol. 40, No. 11-12, 1998, pp. 595-607.

[29] Mahali, P., Acharya, A.A., "Model based test case prioritization using UML activity diagram and evolutionary algorithm", International Journal of Computer Science and Informatics, Vol. 3, No. 2, 2013, pp. 42-47.

[30] K. Ramasamy, and S.A. Sahaaya AruI Mary, "Incorporating varying Requirement Priorities and Costs in Test Case Prioritization for New and Regression testing", Proceedings of International Conference on Computing, Communication and Networking (ICCCN 2008), 2008.

[31] C.R. Panigrahi, and R. Mall, "An approach to prioritize the regression test cases of object-oriented programs", CSI Transaction on ICT Journal, Springer, Vol. 1, No. 2, 2013, pp. 159-173.

[32] P.G. Sapna, and H. Mohanty, "Prioritization of scenarios based on UML activity diagrams", Proceedings of International Conference on Computational Intelligence, Communication Systems and Networks, 2009, pp. 271-276.

[33] M. Weiser, "Program Slices: Formal, Psychological and Practical Investigations of an Automatic Program Abstraction Method", Ph. D. Thesis, University of Michigan, Ann Arbor, MI, 1979.

[34] M. Weiser, "Programmers use slices when debugging", *Communications of the ACM 25*, Vol. 7, 1982, pp. 446–452.

[35] M. Weiser, "Program slicing", *IEEE Transactions on Software Engineering*, Vol. 10, No. 4, 1984, pp. 352–357.

[36] P. Sirivastava, "Test case prioritization", *Journal of Theoretical and Applied Information Technology (JATIT)*, 2009, pp. 178-181.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Table 3: Generated Test Cases From SIG

Test Case No	User No	Books Issued	Book Name	Books after Transaction	Expected Result	Actual Result	Paths Associated
1	16104	-	Operating System	-	Invalid User	Invalid User	Path2
2	13104	5	Computer Graphics	5	Entitled Completed	Entitled Completed	Path3
3	12107	3	Software Testing	3	Error Message	Book Not Available	Path3
4	12406	4	Software Engineering	5	Book Issued Successfully	Book Issued Successfully	Path4
5	11609	2	Data Structure	3	Book Issued Successfully Update	Book Issued Successfully Update	Path5
6	11777	2	Systems Design	NA	Book Not Available	Book Not Available	Path1

Table 5: Cost Of Different Paths

Paths	Path nodes with edges	Edge costs	Node costs	Total path cost	Priority
P1	$1\rightarrow2\rightarrow3\rightarrow11$	5	24	29	V
P2	$1\rightarrow2\rightarrow4\rightarrow5\rightarrow11$	9	32	41	IV
P3	$1\rightarrow2\rightarrow4\rightarrow6\rightarrow7\rightarrow11$	13	38	51	III
P4	$1\rightarrow2\rightarrow4\rightarrow6\rightarrow8\rightarrow10\rightarrow11$	20	42	62	I
P5	$1\rightarrow2\rightarrow4\rightarrow6\rightarrow8\rightarrow9\rightarrow11$	17	42	59	II

Table 6: Prioritized Order Of The Test Cases

Test Cases	Paths Associated	Path Sequence	Priority
T4	Path4	$1\rightarrow2\rightarrow4\rightarrow6\rightarrow8\rightarrow10\rightarrow11$	I
T5	Path5	$1\rightarrow2\rightarrow4\rightarrow6\rightarrow8\rightarrow9\rightarrow11$	II
T3	Path3	$1\rightarrow2\rightarrow4\rightarrow6\rightarrow7\rightarrow11$	III
T2	Path2	$1\rightarrow2\rightarrow4\rightarrow5\rightarrow11$	IV
T1	Path1	$1\rightarrow2\rightarrow3\rightarrow11$	V

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

Table 7: Faults Detected By The Test Cases

Test Cases	Faults								
	FT1	FT2	FT3	FT4	FT5	FT6	FT7	FT8	Total
T1	Ĕ	Ĕ						Ĕ	3
T2	Ĕ	Ĕ				Ĕ		Ĕ	4
T3	Ĕ	Ĕ			Ĕ		Ĕ		5
T4	Ĕ	Ĕ		Ĕ					3
T5	Ĕ	Ĕ	Ĕ			Ĕ		Ĕ	5

Table 8: Increase In APFD % For The Prioritized And Non-Prioritized Test Cases

Sl. No.	Name of the case study	Lines of Code in test cases of original test suite	Number of test cases in the original test suite	Number of faults detected	% of increase in APFD
1	ATM Withdrawal System (ATMWS)	677	26	42	14.27
2	Book Issue of Library Information System (BILIS)	1106	63	61	12
3	User Login System (ULS)	478	24	25	11.36
4	Online Shopping System (OSS)	1426	86	69	12.54
5	Online Ticket Reservation System (OTRS)	903	47	52	13.56