Turkish Journal of Electrical Engineering & Computer Sciences

http://journals.tubitak.gov.tr/elektrik/

Research Article

Turk J Elec Eng & Comp Sci
(2019) 27: 3582 – 3598

doi:10.3906/elk-1806-177

Test case prioritization and distributed testing of object-oriented program

Vipin KUMAR K S[1,*], **Sheena MATHEW**[2]

[1]Department of Computer Science and Engineering, Government Engineering College, Thrissur,India

[2]Department of Computer Science and Engineering, School of Engineering, Cochin University of Science and Technology, Kochi, India

Received: 24.06.2018 • **Accepted/Published Online:** 05.05.2019 • **Final Version:** 18.09.2019

Abstract: Software systems have increased in size and complexity. As a result, object-oriented programming (OOP) is increasingly being used in the development of such large and complex systems. Traditional procedural programming requires a design method that follows a sequential flow of control, which is difficult to follow in the case of large systems design. Thinking in terms of real-life objects and their interactions makes design easier in the case of OOP. However, OOP comes with its own set of disadvantages and testing is one of them. Testing of such systems requires much more effort and time. In our approach the program is analyzed to build a system dependence graph-based model. This model is analyzed for test script generation and to find state-save points within the program where the state of the program can be saved and reused for executing another test case. The test cases are prioritized looking at the structural complexity of the program. The distributed testing presented here is different from traditional distributed testing, where they address testing of web and distributed applications. The distributed architecture developed in this work enables testing of OOP in an efficient manner. The concept is implemented for Java programs as there is support for continuation.

Key words: Distributed testing, system dependence graph, test case prioritization, test script, object-oriented program testing

1. Introduction

Testing of software is essential to ensure reliable usage of software. At the same time there are a number of difficulties attributed to testing of object-oriented programs [1, 2]. Testing of dynamic interactions was discussed with respect to message-method binding in [3]. Testing based on object-state diagrams helps in testing dynamic behavior of the program [4]. Testing usually consumes 20 to 40 percent of the software development life cycle as well as 20 to 40 percent of development cost. A complete overview of software development life cycle (SDLC) and the importance of testing was presented in [5]. That source also presents an introduction to software metrics and its importance. Although object-oriented programming (OOP) helps in easier design and development of large systems, testing of these programs is relatively more time-consuming. It is in this respect that we have developed an architecture for efficient testing of OOP through distribution of test case execution where test cases are prepared looking at the structure of the program under testing. It is worth noting that the survey conducted by Catal and Mishra [6], which reported work from over 100 publication, stressed the need of using public datasets and model-based approaches to testing. In this work we have used a public dataset along with a model-based approach to prioritize test cases and distribute testing of OOP. The test case generation method

*Correspondence: vipin.kumar.k.s@gectcr.ac.in

based on structural analysis of the program provides vital information regarding the test path being tested by each test case. In this work we have developed a compiler-based test case generator, which produces test scripts as output based on priority. The preprocessed program is tested utilizing a distributed architecture, which optimizes the test case execution effort by saving states of the partially executed programs [7]. Processing of the program for priority-based test case generation and preparing the program for distribution through profiling is achieved by following a compiler-based approach. The tool for this purpose is built using ANTLR [8]. A distributed regression testing architecture was discussed in our earlier work [9]. The approach is extended to test object-oriented programs by taking advantage of structure-based test case generation that enables us to distribute the program during testing. The work can be partitioned into the following steps:

1. System dependence graph (SDG)-based model creation for the program.
2. Program structure-based test case generation by analyzing the model.
3. Preprocessing of OOP:
   (a) Analysis of the model for finding state-save points.
   (b) Profiling of code for saving state of the program using continuation and object serialization.
4. Distributed testing and test result aggregation (bring together test subresults).

A model is constructed by extending SDG with control flow. The model is a statement-level representation of the program. The authors of [10] reported the importance of statement level compared to function level and other coarser level representations. Fine level representations like statement level are important for effectiveness and efficiency. It was also reported that fault proneness improves the effectiveness of prioritization. The program is analyzed by the compiler-based tool to construct the extended model based on SDG, which is called Ext-SDG in this work. The model is then analyzed by our compiler-based tool for generating test scripts representing the test cases.

A priority value is calculated based on software metrics for each test script, which is used to rank the test case. The test data are then generated manually for each test script. The compiler-based tool also preprocesses the program to find state-save points within the program so as to profile the code for saving of state using continuation and serialization features of Java. Once this is done, the program is tested using a distributed architecture. The developed program is analyzed using the compiler-based tool (Test-Case Generation and Distributed Testing Tool - TGDT-Tool) developed using ANTLR. The EBNF syntax specification for the input program's programming language is augmented by embedding actions, which constructs the model and analyzes it. These actions create the nodes and edges of the Ext-SDG as well as carrying out analysis-related functions using the data structures and functionalities defined in the grammar file. The test cases and profiled code are generated as part of the analysis using the Ext-SDG. The grammar file for Java is augmented with actions for the creation of TGDT-Tool.

Figure 1 shows the construction of TGDT-Tool using ANTLR. TGDT-Tool is given as output by ANTLR. TGDT is a program in the particular programming language, which is selected as the target in the corresponding grammar file. The tool analyzes the program and constructs the model. It also generates prioritized test cases and profiles the code for distribution.

Figure 2 shows the analysis of the developed program and subsequent model construction for test case generation and profiling of code for distributed testing. The program is given as input to TGDT-Tool, which analyzes the program and constructs an Ext-SDG model for the program. This model is then used to create prioritized test cases and profiling of the code for distributed testing. The profiled code is then executed using

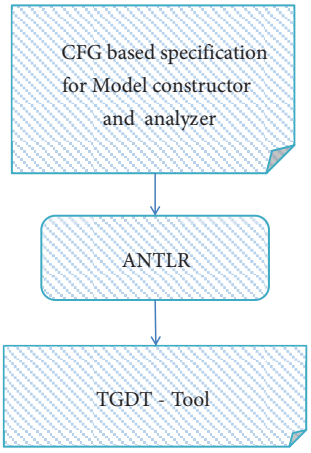

Figure 1. Schematic representation of TGDT-Tool construction.

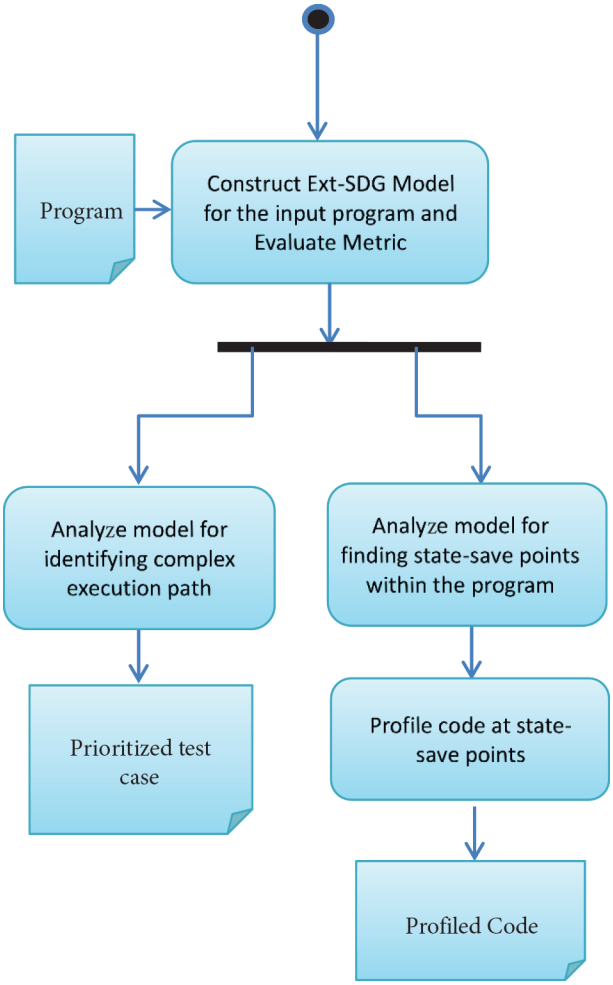

Figure 2. Representation of TGDT-Tool.

a client-server model where a client hypervisor runs on each client to execute the test case when the server distributes the execution of test cases.

2. SDG-based model for OOP

Model-based testing (MBT) is a front-line area of research in software engineering and is continuously evolving. Models are being used in test case generation extensively. Strategies for MBT can be analyzed with respect to various features like model representation, test case generation strategy, coverage, or cost. An extensive comparison of tools based on models used for test case generation was reported in [11]. Use case diagrams and sequence diagrams were used for test case generation in [12]. Many models have been proposed for OOP [13], which captures various features of the program. Each model represents various object-oriented concepts and features. Some models may provide more precise representation when compared to others. The model used in this work is an augmented model. Various models have been combined to provide an efficient model for identifying test cases and finding state-save points within the program. The model is created by extending the SDG [14] with control flow. Liang et al. proposed the class dependence graph and SDG for OPP [14].

Figure 4 shows the Extended-SDG (Ext-SDG) for the program given in Figure 3.

```
CE1   class Calculator
      {
E2          public int add(int a, int b)
            {
S3                int c;
S4                c= a+b;
S5                return c;
            }
      }
CE6   public class CalculatorMain
      {
E7          public static void main(String[] args)
            {
S8                int sum=0;
S9                int i=1;
S10               Calculator c= new Calculator();
S11               while(i<11){
S12                     sum = c.add(sum,i);
S13                     i=c.add(i,1);
                  }
S14         System.out.println(sum  + "\n");
S15         System.out.println(i  + "\n");
            }
      }
```

Figure 3. A sample program.

3. Structure-based test case generation by analyzing the model

The test cases are generated in the form of test scripts representing the test path. Test data are then designed based on these test scripts. The assignment of priority to each test case enables us to test paths from most important to least important ones in that order. This is extremely useful when testing is done in an environment with strict time constraints. Repeated execution of the same statements is eliminated in this approach to achieve faster and efficient execution of test cases during testing. Whenever there is a predicate statement 'c' that branches the execution path our aim is to find a statement 'p' above statement 'c' in the program where the state of the program can be saved and run at other nodes in the network for each alternate path that arise at 'c'. In this approach test cases with the same set of statements at the beginning of execution will be given the same input up to a point 'p' where there are no data-dependence relations from point 'p' to any other statement below 'p'. The test cases in the test suite provide program execution trace information. The test case's execution trace information helps in tracking coverage obtained during testing. The test case also contains other information like test input and expected output along with priority for each test case. Since the execution trace will comprise a large set of statements, we represent the trace as a sequence of critical nodes.

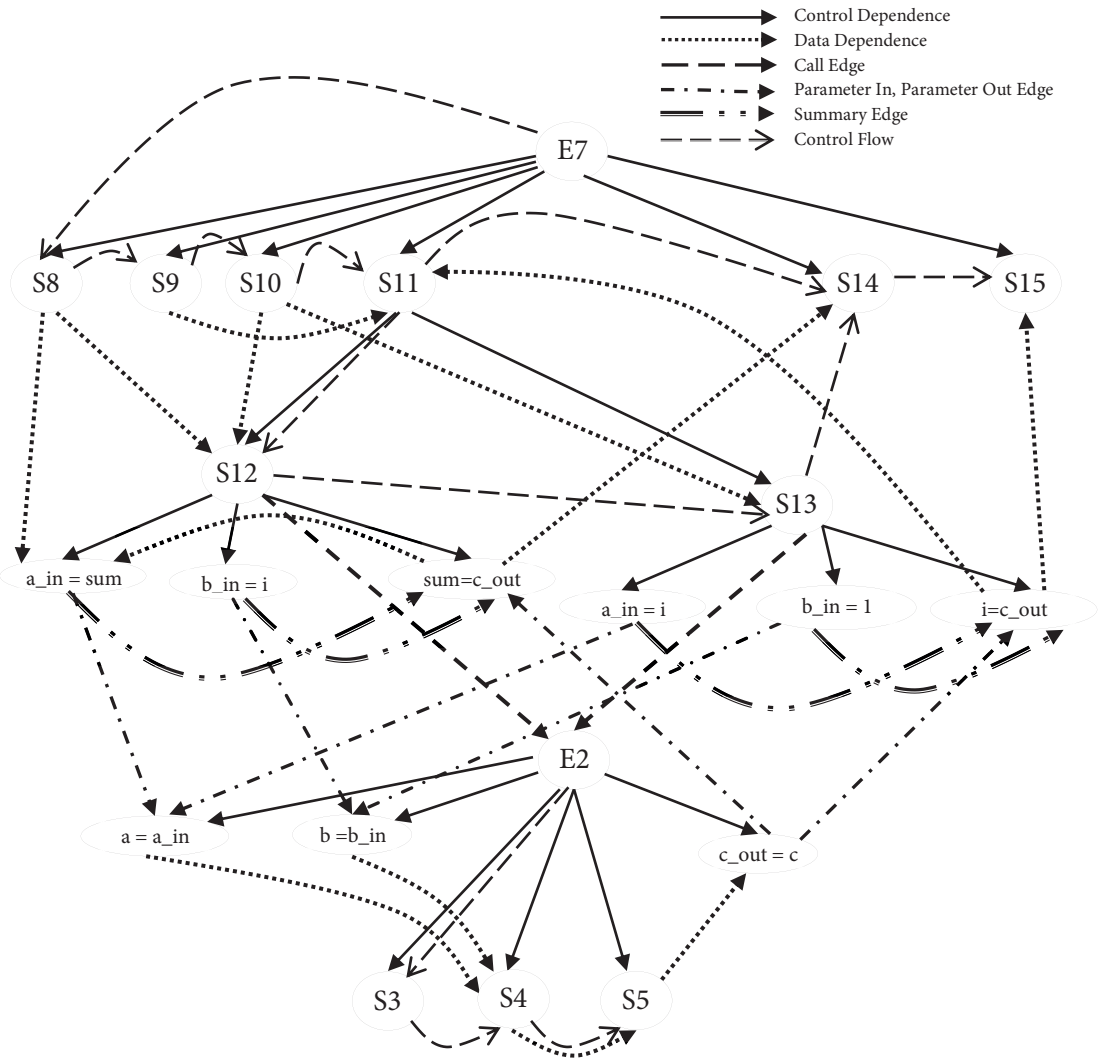

Figure 4. Extended SDG for sample program in Figure 3.

An edge 'e' represented by an ordered pair < ni,nj > is called a critical edge connecting nodes ni and nj if ni is a predicate node (branching node) and nj is the next node along control flow paths originating at ni. Apart from this, the ordered pair < entry node of a method, first node of that method > is also a critical edge.

3.1. Assigning priority to test cases

Test case prioritization helps in efficient selection of test cases. Many approaches have made use of UML diagrams for prioritized test case generation. Preeti et al. [15] discussed an approach for prioritizing test scenarios based on UML activity diagram. Test case prioritization is a very important aspect of regression testing. An extensive review of test case prioritization is available in [16]. Khatibsyarbini et al. [17] reported some of the most pioneering work in this field. A listing of work according to approach used is available in their work. Test case prioritization methods reported in most works either consider coverage or cost as criteria for prioritization [18]. Test case prioritization and optimization based on ant colony optimization technique was discussed in [19]. Priority of a test case is calculated based on the value of the metrics calculated for the

corresponding test case. The priority is a value that reflects the error proneness and complexity of the test path covered by the test case. A path that is more complex than others is assigned a higher priority value. The Ext-SDG is analyzed to identify the paths and the metric values are used to calculate the priority value.

Complexity of a path is considered as a fraction of total complexity:

$$LOC\_Complexity = \frac{Lines\ of\ code\ in\ the\ path}{Total\ lines\ of\ code\ in\ the\ program}$$

For the purpose of prioritization of test cases we have calculated priority based on software metrics, which are important for fault prediction. The authors of [20] reported that almost 11% of work done on test case prioritization for regression testing is fault-based. Numerous metrics have been proposed in the literature for OOP [21]. An extensive reporting of metrics is available in [22, 23]. The compiler-based tool evaluates the values of the metric by analyzing the input program and uses it to assign priority values to each test case. Eleven software metrics were selected from a list of 94 metrics by analyzing the PROMISE dataset [24]. First we removed columns that showed no variation in values, e.g., $ACCESS\_TO\_PUB\_DATA$, as well as metrics representing minimum, average, and maximum values. The metrics representing aggregate total values for the metrics were retained. The minimum, average, and maximum metrics were removed as these values are specific to methods while total values represent class-level information.

The metric list was simplified by first finding metrics that showed high correlation. Then recursive feature elimination (RFE) was used to eliminate metrics further. The result from R Studio obtained after recursive feature elimination is shown in Figure 5. Figure 6 shows the accuracy achieved with respect to number of metrics. The eleven metrics that were selected provide accuracy of over 75 percent.

Variables	Accuracy	Kappa	AccuracySD	KappaSD	Selected
1	0.6518	0.3033	0.18582	0.3713	
2	0.6314	0.2662	0.10169	0.2031	
3	0.6825	0.3660	0.09814	0.1934	
4	0.7136	0.4290	0.13424	0.2674	
5	0.7527	0.5076	0.15739	0.3145	
6	0.7527	0.5076	0.14257	0.2848	
7	0.7627	0.5276	0.15684	0.3132	
8	0.7518	0.5045	0.15170	0.3038	
9	0.7518	0.5045	0.15170	0.3038	
10	0.7718	0.5445	0.14996	0.3002	
11	0.7818	0.5645	0.15529	0.3108	

The top 5 variables (out of 11):
sumLOC_EXECUTABLE, COUPLING_BETWEEN_OBJECTS, sumCYCLOMATIC_COMPLEXITY, DEPTH, FAN_IN

```
> # list the chosen features
> predictors(results)
 [1] "sumLOC_EXECUTABLE","COUPLING_BETWEEN_OBJECTS"  "sumCYCLOMATIC_COMPLEXITY"
 [4] "DEPTH", "FAN_IN",  "WEIGHTED_METHODS_PER_CLASS"
 [7] "RESPONSE_FOR_CLASS","DEP_ON_CHILD",          "LACK_OF_COHESION_OF_METHODS"
[10] "NUM_OF_CHILDREN"          "PERCENT_PUB_DATA"
```

Figure 5. Output obtained from R.

The importance of the selected metrics is shown in Figure 7. The final list of metrics that we retained includes PERCENT_PUB_DATA, COUPLING_BETWEEN_OBJECTS, NUM_OF_CHILDREN, DEP_ON_CHILD, RESPONSE_FOR_CLASS, WEIGHTED_METHODS_PER_CLASS, FAN_IN, sumCYCLOMATIC_COMPLEXITY, LACK_OF_COHESION_OF_METHODS, DEPTH, and sumLOC_EXECUTABLE. The details of these metrics are as follows:

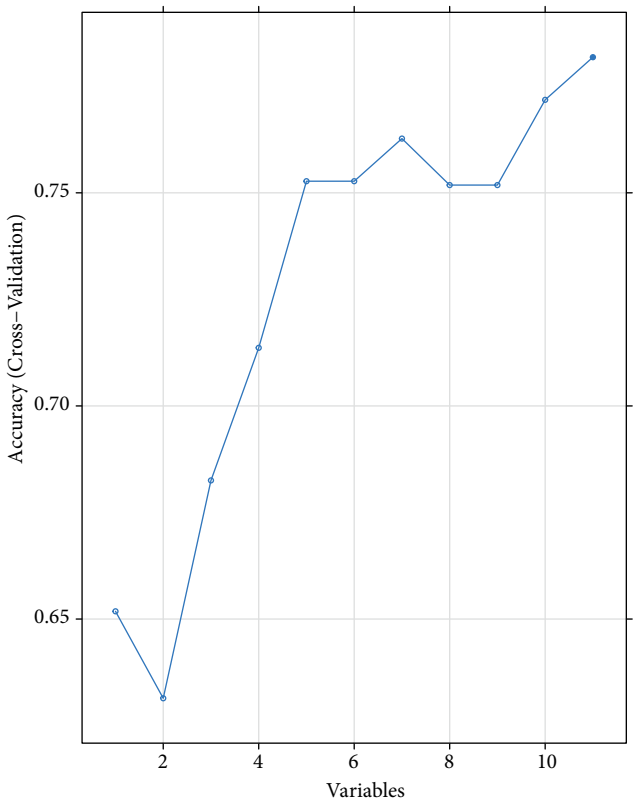

Figure 6. Accuracy with respect to number of metrics.

1. PERCENT_PUB_DATA: This is a measure of the amount of data that publically accessible. It is represented as percentage of data that is public and protected within a class. A low value indicates greater encapsulation.

2. COUPLING_BETWEEN_OBJECTS (CBO): This is a measure of the dependence of a class on other classes. It is the number of distinct classes, based on type, on which a class depends. Classes are considered to be distinct if there is no inheritance relation. CBO for a class is the number of other classes to which it is coupled. Two classes are coupled if methods of a class use methods or instance variables of the other class. Excessive coupling increases the complexity of the software and affects reliability.

3. NUM_OF_CHILDREN: The number of classes derived from a specified class.

4. DEP_ON_CHILD: Whether a class is dependent on a descendant.

5. RESPONSE_FOR_CLASS: This is total number of methods implemented as well as accessible to an object class due to inheritance.

6. WEIGHTED_METHODS_PER_CLASS: This is the total number of methods implemented within a class excluding those inherited as such.

7. FAN_IN: This is a count of other methods that call a particular method. Usually modules in the higher levels of design structure call modules in a lower level.

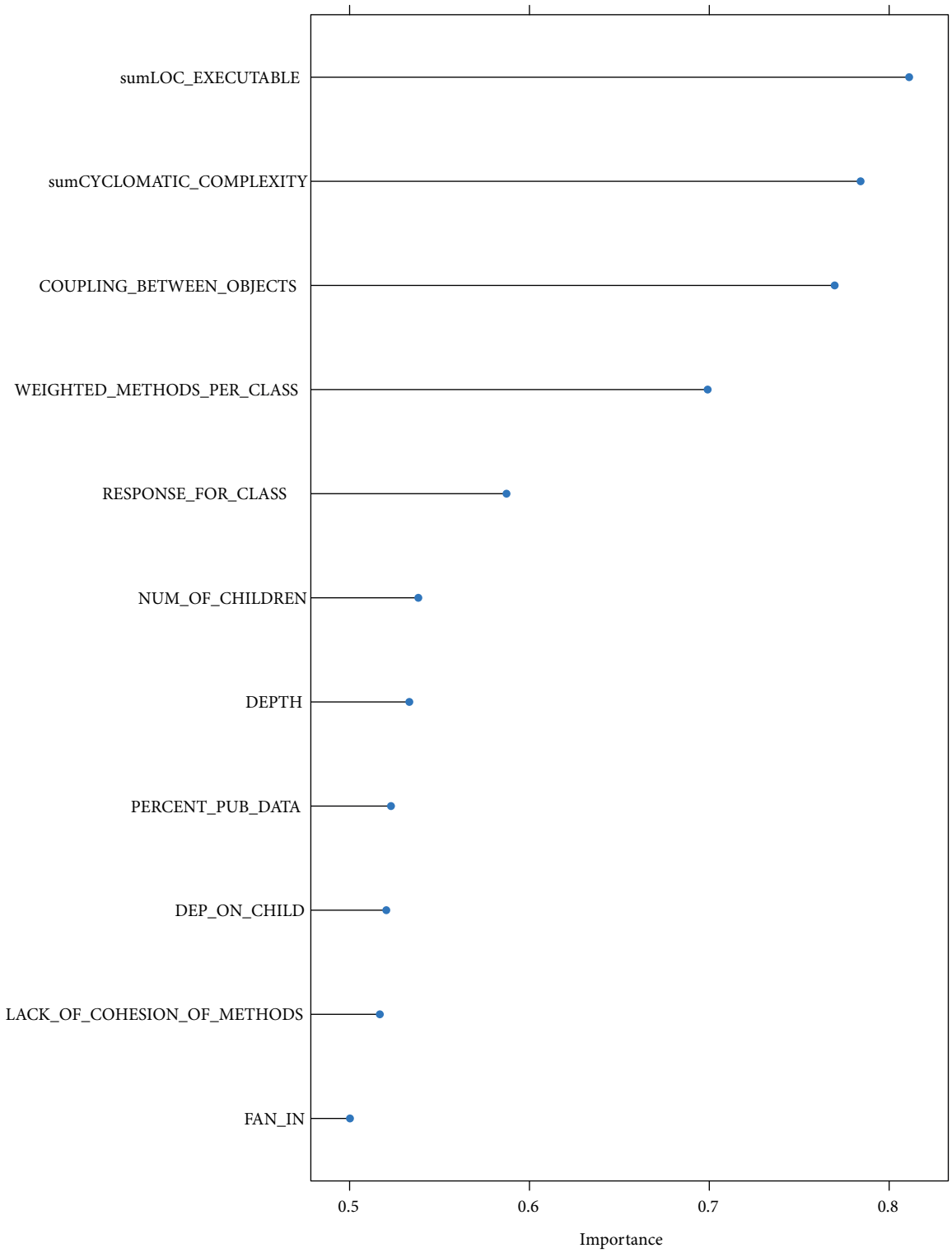

Figure 7. Importance calculated for each metric.

8. sumCYCLOMATIC_COMPLEXITY: It is the measure of the complexity of the decision structure of a method or a program. It is represented as a count of linearly independent paths within a method

or program. The count represents the total number of different paths that need to be tested for path coverage. The metric was proposed by McCabe and gained wide acceptance. For a control flow graph G with n nodes and e edges the cyclomatic complexity v(G) is computed in the following two ways:

(a) v(G) = e – n + 2,

(b) v(G) = number of decision nodes + 1.

9. LACK_OF_COHESION_OF_METHODS: For each data field in a class, the percentage of the methods in the class using that data field; the percentages are averaged and then subtracted from 100. The locm metric indicates low or high percentage of cohesion. If the percentage is low, the class is cohesive. If it is high, it may indicate that the class could be split into separate classes that will individually have greater cohesion.

10. DEPTH_OF_INHERITANCE_TREE (DEPTH): This metric represents the level for a class in the inheritance hierarchy. For instance, if a parent has one child the depth for the child is two. Depth indicates at what level a class is located within its class hierarchy. In general, inheritance increases when depth increases. As the depth increases, it becomes more and more complex to predict the behavior of the class. It is preferable that DEPTH be less than or equal to 4.

11. sumLOC_EXECUTABLE: Source lines of code that contain only code.

The importances calculated for these metrics are as follows: sumLOC_EXECUTABLE - 0.8111, sumCYCLOMATIC_COMPLEXITY - 0.7841, COUPLING_BETWEEN_OBJECTS - 0.7697, WEIGHTED_METHODS_PER_CLASS - 0.6991, RESPONSE_FOR_CLASS - 0.5872, NUM_OF_CHILDREN - 0.5382, DEPTH - 0.5332 ,PERCENT_PUB_DATA - 0.5230, DEP_ON_CHILD - 0.5204, LACK_OF_COHESION_OF_METHODS - 0.5168, FAN_IN - 0.5002.

The priority values reflecting the complexity and error proneness of each path are assigned by proportionately allocating the total metric value among each test paths. For 'n' number of metrics the equation for priority for test cases is as follows:

$$Priority\ for\ a\ test\ case\ tc_j = \sum_{i=1}^{n} \frac{(Value\ of\ m_i\ for\ test\ case\ tc_j)}{(Value\ of\ m_i\ for\ the\ program\ P} \times Importance\ of\ m_i$$

The total of each metric value assigned to a test case reflects its priority. A test case with the highest value is considered to be the one having the highest priority and the one with the lowest value is considered to have the least priority. The priority is assigned to each path by recursively descending through each path and calculating the priority as we backtrack through each path. Each path is identified with the help of critical edges, which identifies each segment in a test path uniquely. The test cases are ranked according to the priority value evaluated for the test case based on the path covered by the test case. Following this the test data are designed and linked with the test case. This enables the tracking of path being covered during each testing.

The algorithm for test script generation is presented in Algorithm. The 'CompressAndCalculate' function compresses the path to include only the critical edges as well as calculating the complexity value for prioritization of test cases.

Algorithm Test script generation algorithm.

```
1: function TestScriptGen(PathSet temp, TestScriptSet final, Node m)   ▷ temp - stores
   the test path until a termination is encountered, final - once a termination node is encountered the path is
   moved to final after compression and evaluation of priority value, m is a node of the Ext-SDG
2:     for all Node n from left to right adjacent to m in CFG do
3:         if if n is not a visited node then
4:             mark n as visited
5:             if  n is a termination node  then
6:                 for  all paths p in temp that end in m  do
7:                     set p' to CompressAndCalculate(addnode (p,n))
8:                     add p' to final
9:                 end for
10:            else
11:                for all paths p in temp that end in m  do
12:                    remove p from temp
13:                    add path p', obtained by appending n to p, to temp
14:                    TestScriptGen(temp, final, n)
15:                end for
16:            end if
17:        end if
18:        for all paths p' in temp that end in n do
19:            remove p' from temp
20:            add path p, obtained by removing n from p', to temp
21:        end for
22:        mark n as unvisited
23:    end for
24: end function
```

3.1.1. Test case representation

The sequence of critical edges provides complete information about an execution trace. The critical edges comprising an execution trace are stored in the form of a linked list where nodes of the linked list are of the form <startNode;endNode;link>. A logical view of the test case is presented in Figure 8.

4. Preprocessing of OOP

The input program needs to be profiled by inserting code that enables distributed execution of the program. The test cases are split accordingly to allow distributed execution at different nodes. For distribution of execution, the code is preprocessed to find the state-save points within the program. Then the code is profiled at these points by adding code to save the dynamic state of the program during runtime. The saved state is then distributed in a distributed environment and executed. The profiled code is tested by distributing across different nodes.

4.1. Analysis of the model for finding state-save points

The SDG is analyzed to find points within the program that will enable saving of the state of the program. Every statement that leads to alternation in the control flow within a program provides an opportunity to distribute the execution through saving of the state during runtime. The program is analyzed with respect to this point to find a point within the program where the program state can be saved. The point is selected such that neither the alternation statement nor any other statement that lies after this statement has any data dependence with any other statement above this point (Figure 9). Java allows programs to be frozen and then

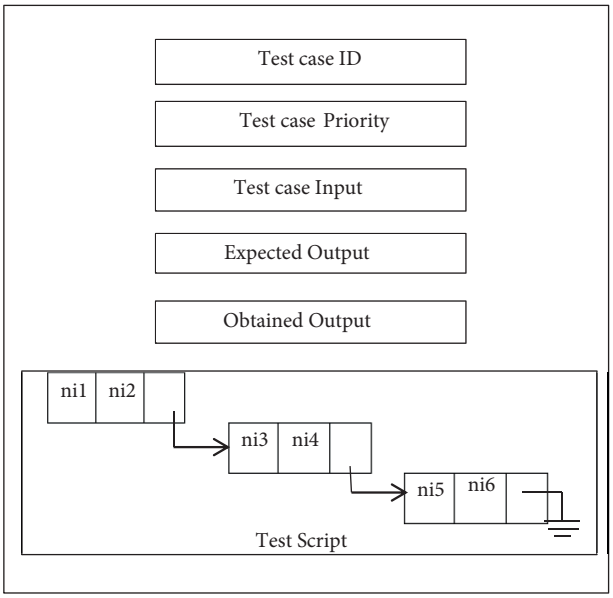

Figure 8. Test case representation.

the state to be saved and distributed. The test case along with the saved state of the program is passed to the node for execution. Information regarding test case ID and point from where the test case needs to be executed is made available at the node.

4.2. Profiling of code for saving state of the program

Code is profiled at the points identified by data dependence analysis by inserting code that enables the freezing of program state and then serializing it to enable distribution in a distributed execution environment. The points for profiling the code are identified by automated analysis of the program. As the code inserted into the program does not have any dependence and no other part of the program has any dependence on the inserted code, it does not affect the processing or output generated by the program. In that sense, the profiled code is equivalent to the original code.

Figure 9 depicts the identification of state-save points for profiling the code by data dependence analysis. The code is profiled at these points so as to be able to distribute the execution and testing. The part of the TGDT-Tool that profiles the code is shown in Figure 2.

5. Distributed testing and test result aggregation

Testing of distributed systems has a different set of challenges attributed to it, like size of application, synchronization, message passing, and race condition. Issues related to testing large distributed systems were dealt with in [25]. Client-server-based architecture has been used extensively in testing distributed systems. However, testing of OOP using a distributed architecture is a different problem. In this work we have used a distributed architecture for testing OOP. Distributed architecture is client-server-based system with a client hypervisor running on the client, which upon receiving the saved states from the server reinstantiates the state of the program to continue from the point where the state was saved. The client saves the state of the program and serializes it. This is sent to the server, which again schedules it for execution on any other client node. The first client continues the execution of the test case and upon completion of the execution the client hypervisor updates the status of the client.

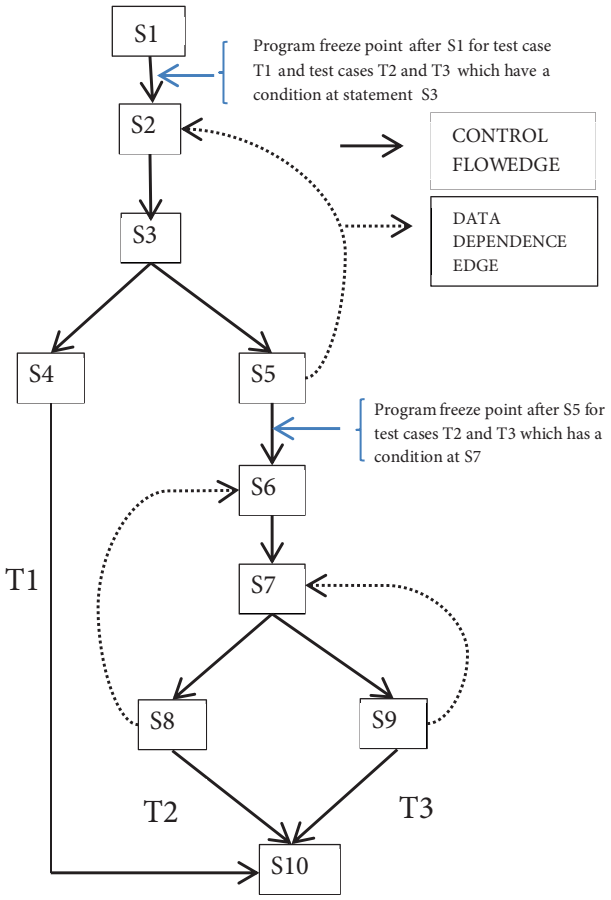

Figure 9. Identifying state-save points.

Figure 10 shows the structure of the client hypervisor. The clients communicate with the server and the corresponding system is represented in Figure 10. As shown in Figure 9, when test case T1 is executed, the state is saved and distributed during the execution corresponding to test case T1. Test case T1 continues

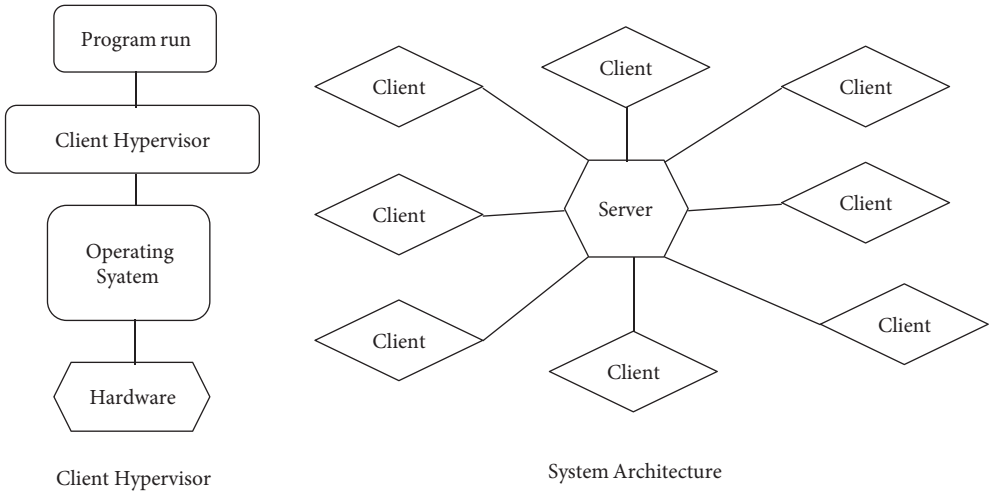

Figure 10. System architecture.

after sending the saved state to the server. The state is reestablished to execute test case T2. As test case T2 executes, the state is saved and distributed for executing test case T3. The test case runs corresponding to the arbitrary program in Figure 9 as depicted in Figure 11.

R1	S1				
T1	R1	S2	S3	S4	S10
R2	R1	S2	S3	S5	
T2	R2	S6	S7	S8	S10
T3	R2	S6	S7	S9	S10

Figure 11. Test case executions corresponding to the arbitrary executions in Figure 9.

The results of the execution need to be aggregated as a single test case's execution may span different client nodes. The output generated at each client node is aggregated as shown in Figure 12.

OUTPUT_ R1	OUTPUT TILL STATEMENT S1				
T1	OUTPUT_ R1	OUTPUT OF S2,S3,S9			
OUTPUT_R2	OUTPUT_ R1	OUTPUT OF S2,S3,S5			
T2	OUTPUT_ R2	S6	S7	S8	S10
T3	OUTPUT_ R2	S6	S7	S9	S10

Figure 12. Test case result aggregation.

6. Performance evaluation

The performance was analyzed during several testing rounds. The total time consumed during the execution of the test cases during two different scenarios is reported here. Measurements were recorded with a precision of one by ten millionths of a second. Figure 13 shows the time taken in total during several testing rounds in two scenarios.

The first scenario corresponds to a testing environment where testing is done on a distributed architecture without saving of state. Such an architecture is widely used for testing software. During testing the total number of test cases in the test suite is split among different nodes in the system. The allocated test cases are run on individual machines and the result is then aggregated at a later stage. For measuring the execution time, the individual times are calculated and are added to get the total execution time of a machine. The maximum of the total execution times of the machines is taken as the time required for the method. A call to nanoTime() was made at the start and end of the main method. The difference between these times is taken as the execution time.

The second scenario corresponds to this work, which incorporates saving of state. The state of the program during execution is saved and distributed for execution of another test case. Figure 13 shows the result

of execution. The time is recorded in scenarios where execution is done without saving of states and with saving of states. In the scenario where execution is done without saving of state, a similar architecture was used, but instead of saving the state each node executed the test case completely.

Without state saving	With state saving
871.7109973	568.5983791

Figure 13. Time taken during several testing rounds for the two scenarios.

The execution of test cases with saving of state consumed only 65 percent of the time required for executing test cases without saving of state.

Figure 14 shows a comparison of time taken during both scenarios. However, there is scope for further improvement. Once the test cases are selected for execution based on priority, the test cases are executed according to FCFS scheduling. If execution of test cases is done by taking into account execution time, further improvement is possible.

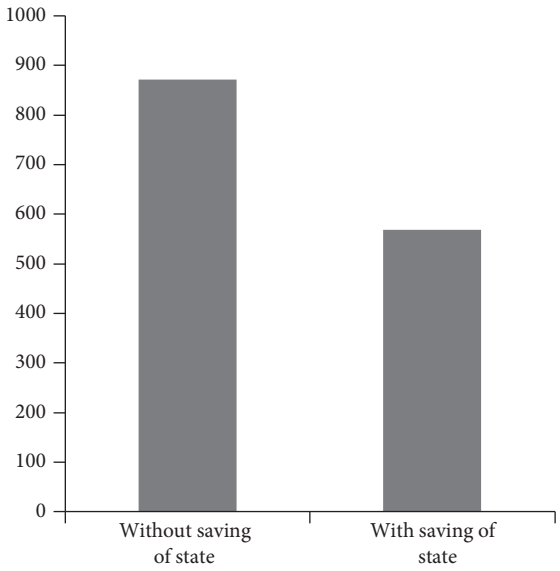

Figure 14. Comparison of test case execution during two scenarios.

7. Conclusion

The approach used in this work is based on a fault-based approach using metrics. The metrics were selected systematically following feature selection through elimination. Previous data relating to fault occurrence from a public dataset were used for this purpose. Eleven metrics were used depending on accuracy level obtained. The work in [26] estimates code complexity based on statement count. The number of decision nodes is multiplied by 5 and added to the count of nondecision statements in the program to represent code complexity. The metrics were selected in this work from among 94 metrics looking at the error proneness of each metric. The work in [27] reported prioritization of test cases using a call graph for refactoring errors. The SDG-based approach utilizing error proneness features used in this work is a unique approach. The statement level representation

in SDG allows us to evaluate and store these metric values at suitable nodes in the Ext-SDG, which is used to prioritize test cases during path-based analysis of the model. There has been much work related to test case prioritization using different approaches. The proposed system for testing is a distributed testing mechanism with saving of state. The approach to run the test cases during initial testing with test cases generated for the developed program utilizing a distributed architecture with saving of states is a novel one. Prioritized test cases generated following a path-based analysis are executed in a distributed manner with saving of states. For this, the program is profiled at points suitable for saving of states, by inserting code that saves the state of the program at that point and distributes the execution to another node where the same state is reinstantiated. Execution of another test case is taken up at this node, which is not affected by the execution done so far. Thus, saving of state at potential points results in saving of time by eliminating repeated execution of code. The proposed method was implemented for Java programs and the performance is compared with existing systems. It was found that the system required only 65 percent of the time when compared to distributed testing without saving of state. Furthermore, this helps to prioritize test cases, which helps in faster revealing of errors.

Sl. No.	Criteria	Explanation
1	Validity of model	The model used here is Ext -SDG which is based on SDG. The basic SDG is extended with control flow for path based analysis. The Ext-SDG model is capable of representing object -oriented features like class, object, methods, method invocations, control flow, inheritance hierarchy and polymorphism along with control dependence and data dependence which are important to identify state save points within the program. Data dependence edges in SDG represent def -use within the program explicitly. SDG allows statement level representation which allows analysis to be done at variable level which is important to identify state save points with in the program.
2	Test Case Generation strategy	Test cases are generated by analyzing paths. The eleven software metrics were selected from ninety four metrics by feature elimination techniques which reflect on the error proneness of the test cases. The test scripts representing the test cases were generated to cover all paths which guarantee exercising of all linearly independent paths.
3	Test Data generation	Test data generation was done manually looking at state save points along independent segments .
4	Usability	Test cases are prioritized in such way that they most likely reveal maximum number of errors. The distributed architecture helps in testing OOP efficiently.
5	Testing level	The architecture is used at system level testing of OOPs. The developed program's SDG is analyzed to identify test cases that execute complex paths which are system level test cases.
6	Applicability to Regression testing	The model is suitable for regression testing. The trace information helps in identifying test ca ses that needs to be rerun.

Figure 15. Summary of evaluation criteria for the proposed system.

Figure 15 provides a summary of the work with respect to some important criteria for evaluation of the test case generation strategy.

7.1. Future work

1. This implementation can be extended to other object-oriented language. Some modern languages have built-in facility to capture program state using continuations.

2. The approach to prioritization of test cases can be extended to method-level testing, class-level testing, and regression testing of OOP.

3. The proposed system implements a raw FIFO scheduling algorithm. It can be optimized for better performance.

4. Shared memory implementation can be used to store and retrieve test cases, which will reduce the network traffic and improve efficiency.

5. The model is subjected to static analysis as a result of dynamic behavior of the system like dynamic flow of control, dynamic polymorphic, and the like, which has not been analyzed.

References

[1] Morris S, Johnson J. A survey of testing techniques for object-oriented systems. In: Conference of the Centre for Advanced Studies on Collaborative Research; Toronto, Canada; 1996. pp. 17-24.

[2] Jorgensen PC, Erickson C. Object-oriented integration testing. Communications of ACM 1994; 37 (9): 30-38. doi: 10.1145/182987.182989

[3] Zhao R, Lin L. An UML statechart diagram-based MM-path generation approach for object-oriented integration testing. International Journal of Applied Mathematics and Computer Sciences 2008; 2 (10): 3470-3475.

[4] Gao J, Kung D, Hsia P, Toyoshima Y, Chen C. Object state testing for object-oriented programs. In: IEEE Computer Software and Applications Conference; Dallas, TX, USA; 1995. pp. 76-85.

[5] Mall R. Fundamentals of Software Engineering. 3rd ed. New Delhi, India: Prentice Hall of India, 2010.

[6] Catal C, Mishra D. Test case prioritization: a systematic mapping study. Software Quality Journal 2013; 21 (3): 445-478. doi: 10.1007/s11219-012-9181-z

[7] Vipin KKS, Sheena M. Model based distributed testing of object oriented programs. In: Elsevier 2014 Procedia Technology - International Conference on Information and Communication Technologies; Kochi, India; 2014. pp. 857-866.

[8] Terence P. Definitive ANTLR Reference: Building Domain Specific Languages. Raleigh, NC, USA: Pragmatic Bookshelf, 2007.

[9] Vipin KKS, Lallu A, Sheena M. An efficient approach for distributed regression testing of object oriented programs. In: ACM International Conference on Interdisciplinary Advances in Applied Computing; Amritapuri, India; 2014. pp. 331-337.

[10] Elbaum S, Malishevsky AG, Rothermel G. Test case prioritization: a family of empirical studies. IEEE Transactions on Software Engineering 2002; 28 (2): 159-182. doi: 10.1109/32.988497

[11] Sarma M, Murthy PVR, Jell S, Ulrich A. Model-based testing in industry: a case study with two MBT tools. In: ACM Fifth Workshop on Automation of Software Test; New York, NY, USA; 2010. pp. 87-90.

[12] Swain SK, Mohapatra DP, Mall R. Test case generation based on use case and sequence diagram. International Journal of Software Engineering 2010; 3 (2): 21–52.

[13] Larsen L, Harrold MJ. Slicing object-oriented software. In: IEEE International Conference On Software Engineering; Washington, DC, USA; 1996. pp. 495-505.

[14] Liang D, Harrold MJ. Slicing objects using system dependence graph. In: IEEE International Conference on Software Maintenance; Bethesda, MD, USA; 1998. pp. 358-367.

[15] Kaur P, Bansal P, Sibal R. Prioritization of test scenarios derived from UML activity diagram using path complexity. In: ACM CUBE International Information Technology Conference; Pune, India; 2012. pp. 355-359.

[16] Kiran RS, Chandraprakash, Srinivas K. A literature survey on TCP-test case prioritization using the RT-regression techniques. Global Journal of Research In Engineering 2015; 15 (1): 29-40.

[17] Khatibsyarbini M, Isa MA, Jawawi DNA, Tumeng R. Test case prioritization approaches in regression testing: a systematic literature review. Information and Software Technology 2018; 93: 74-93. doi: 10.1016/j.infsof.2017.08.014

[18] Kumar A, Singh K. A literature survey on test case prioritization. COMPUSOFT International Journal of Advanced Computer Technology 2014; 3 (5): 793-799.

[19] Ansaria A, Anam K, Alisha K ,Mukadam K. Optimized regression test using test case prioritization. Procedia Computer Science 2016; 79: 150–160. doi: 10.1016/j.procs.2016.03.020

[20] Singh Y, Kaur A, Suri B, Singhal S. Systematic literature review on regression test prioritization techniques. Informatica International Journal of Computing and Informatics 2012; 36 (4): 379–408.

[21] Chidamber SR, Kemerer CF. A metric suite for object-oriented design. IEEE Transactions on Software Engineering 1994; 20 (6): 476-493. doi: 10.1109/32.295895

[22] Fenton NE, Neil M. Software metrics: roadmap. In: ICSE Conference on The Future of Software Engineering; Limerick, Ireland; 2000. pp. 357–370.

[23] Xenos M, Stavrinoudis D, Zikouli K, Christodoulakis D. Object-oriented metrics – a survey. In: FESMA 2000, Federation of European Software Measurement Associations; Madrid, Spain; 2000.

[24] Koru AG. The PROMISE Repository of Software Engineering Databases. Ottawa, Canada: School of Information Technology and Engineering, University of Ottawa, 2005.

[25] Almeida EC, Marynowski JE, Sunyé G, Traon YL, Valduriez P. Efficient distributed test architectures for large-scale systems. In: Springer Lecture Notes on Computer Science ICTSS 2010 Testing Software and Systems; Berlin, Germany; 2010. pp. 174-187.

[26] Haraty RA, Mansour N, Moukahal L, Khalil I. Regression test cases prioritization using clustering and code change relevance. International Journal of Software Engineering and Knowledge Engineering 2016; 26 (5): 733–768. doi: 10.1142/S0218194016500248

[27] Alves ELG, Machado PDL, Massoni T, Kim M. Prioritizing test cases for early detection of refactoring faults. Software: Testing, Verification and Reliability 2016; 26 (5): 402–426. doi: 10.1002/stvr.1603