International Journal of Engineering & Technology, 7 (4.1) (2018) 28-32

International Journal of Engineering & Technology

Website: www.sciencepubco.com/index.php/IJET

Research paper

Evolutionary Cost Cognizant Regression Test Prioritization for Object-Oriented Programs Based on Fault Dependency

Abdulkarim Bello*, Abubakar Md Sultan, Abdul Azim Abdul Ghani, Hazura Zulzalil

Faculty of Computer Science and Information Technology, University Putra Malaysia
** Corresponding author E-mail:kiyawa99@yahoo.com:*

Abstract

Regression testing performed to provide confidence on the newly or updated software system is a resource consuming process. To ease this process, various techniques are developed. One such technique, test case prioritization, orders test cases with respect to the goals such that the most important test case in achieving those goals is scheduled earlier during the testing session. Among such performance goals, the rate of faults detections, measure how faults are detected quickly throughout the regression testing process. Improved dependency detection among faults provides faster feedback to the developers which gives chance to debug leading faults earlier in time. One other goal, the rate of fault severity detection, measure how fast severe fault can be detected in the testing process. Although, previous works address these issues but assumed that the costs of executing test cases and severities of detected faults are the same. However, costs of test and severities of faults varied. Furthermore, they did not consider incorporating evolution process such as applying genetic algorithms to their technique. In this work, we proposed an evolutionary cost-cognizant regression testing approach that prioritizes test case according to the rate of severity detection of test cases connected with dependent faults using genetic algorithms. The aim is to reveal more severe leading faults earlier using least cost in executing the test suite and to measure the efficacy of the technique using APFDc.

Keywords: *Test Case Prioritization, Regression testing, Genetic Algorithms, APFDc*.

1. Introduction

Software testing occurs simultaneously while developing the software to detect errors earlier in time and to ascertain that changes made during the software update did not affect the system under development or maintenance negatively.

However, during the development phase, as the test suite is updated and tends to increase in size. Due to the amount of human resource and time constraints needed to re-execute a larger test suite, it is necessary required to develop techniques that will reduce the effort of performing regression testing [1, 2].

Various regression testing techniques have been proposed, both in the industry and academic, to reduce regression testing effort [3-5]: test suite reduction/minimization [6, 7]; selection [8, 9]; and prioritization [10, 11]. Test suite minimization techniques aim at identifying and eliminating redundant test cases from the suite. Test case selection techniques identify a subset of test suite, required to re-test the changes in the software. Test suite prioritization technique schedules test cases in an order that increases the early fault detection [12].

Regression test cases selection techniques [8, 13, 14] chose the most appropriate test cases from the existing test cases such that the costs of performing regression testing are reduced according to the information gathered from the program source code and the changed version or on the program specifications. Test suite minimization techniques attempt to reduce the costs of performing regression testing to have earlier feedback. This technique reduces the test suite to the minimum subset with the same coverage as the original test suite thereby lowering the cost. [5].

These methods, however, may have potential problems. For example, although some researchers argue that it is little or entirely no any loss for a minimized test subset to detect faults [15], another recent empirical study reveals that the fault detection capabilities of test suites can be reduced severely by test suite minimization [16]. Although, some reports safe regression test selection techniques which assure that the selected subset of the test cases have equal fault revealing capability as compared to the original test suite [17], these situations for which safety can be achieved with the minimized test suite may not always be able to hold [10, 18].

One other technique that has the capability of increasing the cost-effectiveness of regression testing is test case prioritization. Test cases prioritization techniques find another style of improving regression testing process. Based on information associated with a test suite, such as coverage information or information estimating test cases' potential ability to reveal faults, test case prioritization techniques can be used for the scheduling of test cases, such that those test cases could be run in a specific order for maximizing some objective function.

For example, testers may want to find faults possibly earlier or increase coverage of the program at the fastest rate possible. Considering smaller size test suite enables all the test cases to be run, simply running all test cases in any order may be more effective [11], while performing test prioritization then running the test cases. However, considering the time needed to run the whole test cases to be long, prioritization techniques adds benefits by sched-

Copyright © 2018 Authors. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

uling the most important test cases to run earlier. For example, suppose that the available time and resource constraints can enable only a part of the test suite to be run. In this case, it may be better to use test case prioritization techniques to cause the most important test cases for achieving the tester's objectives to execute earlier in time, rather than running test cases in a random order.

In this study, a prioritization approach that considers dependencies among faults to prioritize test cases for regression testing was presented. The approach also uses genetic algorithms to searches through the state space and come up with an optimized order based on the objective function.

The remaining part of this paper is organized as follows. Literature related to this work was discussed in section 2. Sections 3 and 4 present materials and methods for the research and discussed the result and section five gives a conclusion.

2. Related Works

Regression testing is a software testing process that checks to finds out that software that was developed and working correctly before still works correctly after it was updated, changed or joint with another software. Regression testing is carried out between two different version of the software to prove that the newly added software components did not introduce to the originally well-working component of that software [1]. Test case prioritization techniques in order to test cases in a way that the effectiveness of the test cases is increased based on the defined requirements.
To gain more insight in to the progress of research in regression testing prioritization process, this section briefly discusses some prioritization techniques according to according to [4]. Although [4] categorized it into eight broad categories, few modifications are added to go with the current trend in the field.
Geol and Sharma [19] consider faults exposing potential to prioritize teste cases. Tus further, developed a regression testing technique that prioritizes test cases based on their exposing potential. Their approach uses total code coverage exercised by each test case to measure the efficacy of the technique. The technique was developed to minimize the time, effort and cost of regression testing. The program on which the technique was evaluated and how the technique was evaluated were not mentioned. Though fault exposing potential was mentioned as a measure of identifying the test case priority. Lou et al [20] proposed test case prioritization for software evolution that considers the selection of test cases according to faults that occur due to mutation and the difference between early and later versions of the software. The technique uses two models to get the sum of all the fault detection capability of the technique. Di Nardo et at [21] proposes a regression testing technique that considers four different coverage measures: total, additional, total coverage of modified code; and additional coverage of modified code. The technique was run on five different versions of an industrial software and APFD metric was used to evaluate the technique.
There is an argument by Kumar and Chauhan in [22] that most test case prioritization techniques did not consider identifying badly affected modules considering the changes that are propagated from one module to other modules. Therefore, proposed a test case prioritization technique that identifies such affected modules and prioritizes test cases using a module dependency matrix and coupling slices. The technique was tested by executing it on a sample program consisting of ten modules and was found to reveal faults considerably better than when not prioritize with little efficacy. Kayes and Chanareski [23] developed a regression test case prioritization technique based on the dependency network of faults. The technique is built based on the fact that dependent faults are a consequence of independent faults can directly be detected and removed. Wang et al [24] proposed a test case prioritization technique based on additional statement coverage to prioritize test cases by considering fault severity as the main factor. The experimental result shows that applying technique can improve the efficiency of regression testing.
Amrita and Kumar [25] proposed an RTP approach for analysis regression testing technique. The technique considers detected faults and the cost to detect the faults for prioritization of test cases. When compared with the random test case prioritization technique, the technique was found to perform efficiently better. Average Percentage of Fault Detection Cost (APFDc) was the metric used for measuring the performance of the new technique. Di Nucci et al in [26] proposed genetic algorithm technique to solve test case prioritization problem when using multiple test criteria after noticing that most techniques used a simplified form of Area under Curve (AUC) metric. Results obtained by an empirical study on six real-word open source programs indicates that the technique is more cost-effective than simple greedy search technique.

3. Materials and Method

In this section, we discussed program representation, Java system dependence graph, slicing technique, our proposed technique and demonstrate how the technique works with some examples.

3.1. Program Representation

One of the fundamental aims was to make program development and maintenance as easy as possible [27], fast and with less prone to error. This adds to it the problem of understanding what an existing software system does and how it operates [15], understanding the vacation between several different version of software programs and creating new programs by putting together different components of the old program components. Some program representation concepts were discussed in the following section.

3.1.1. Control Flow Graph (CFG)

CFG represents the flow of a software program diagrammatically. Can be seen as a diagrammatical representation of program execution. CFG is mostly used in static analysis as well as compiler applications [17], as they can accurately represent the flow inside of a program unit. Is process oriented and can show all the paths that can be traversed during program execution it is also be seen as a directed graph that has edges which represent control flow paths and nodes representing basic blocks.

3.1.2 Program Dependence Graph (PDG)

PDG is a graphical representation of within a single procedure program. This graph is represented by nodes which represent statements in a program and edges representing the dependencies that exist between the statements.

3.1.3. System Dependence Graph (SDG)

SDG extends the program dependence graph to accommodate a collection of procedures instead of presenting just a program containing only one procedure. SDG represents not only dependences, it also represents different other relationships that exist within program source code.

3.1.4. Extended System Dependency Graph (ESDG)

ESDG was presented by [28] as an extension of SDG presented by [29] to model procedural programs. Previously, ESDG was used in the studies [30] to model object-oriented programs to performed regression testing. ESD can be seen as a directed connected graph that represents different kinds of dependencies that exist within a software system that includes both control and data dependencies and dependencies that occurred as result of addition features of the object-oriented construct.

3.1.5. Java System Dependence Graph (JSDG)

A JSDG is a directed graph that depicts both control and data dependencies that exist among the statements within a Java program. The graph represents Java program different individual graph and later maps all the individual graphs in to a single larger graph. Statements are represented by nodes are categorized as either belonging to the structure of a program or represent the behaviour. [31].

Statements

In JSDG, the lowest layer of JSDG representation is the statement layer. It is the smallest construct which represents an expression in a program's source code. There can also be a statement that represents a call to another method thereby requires a special representation [31].

Method Dependence Graph (MDG)

MDG is a directed graph that depicts the representation of a single procedure program. MDG is more like procedure dependence graph in that both represent single method. [32–35]. The method entry vertex is connected to any other vertices belonging to the method via control dependence edges.

Class Dependence Graph (ClDG)

ClDG is an intermediary representation of a class in software programs [36]. Multiple MDGs are combined to form a ClDG. There is one entry vertex any class in ClDG, that is linked to vertices that server as an entry point to methods with the help of a class member edge. The visibility of the memberships can be marked as either public, private, protected or default [37]. Data member edge is used to indicates the relationship where one class inherits features of another class. The data member of a class is linked to the class entry vertex through the data member edges.

Interface Dependence Graph (InDG)

There is one entry vertex for each InDG that provides a connection from the InDG entry vertex to different methods entry vertices that represent the abstract method declared within that interface with the help abstract member edge. Input parameter vertices are connected to every method entry vertex. For all the methods the are implementing the abstract methods of the interface, there entry vertices are connected to the entry vertices of the abstract methods entry vertices through the implement abstract method edges.

Package Dependence Graph

A package holds a number of different classes that are dedicated to carrying out a specific task. A package is represented by a graph called the package dependence graph (PaDG) [37, 38]. In PaDG, the visibility of a class within the package is of highly important such
that the services of a class reach to all other classes can have access to the class. For every class of a package, there is an entry vertex called package member edge that connects virtually the entry vertex of the all classes within that package to the entry vertex of the package.

JSDG Vertices

In JSDG, a vertex represents a statement of a program which can be of four different types based on the function it performs. The four different vertices include the statement vertex, parameter (in/out) vertex, polymorphic vertex and entry vertex.
Statement Vertices: represent all the statements of a method. There are two categories of these vertices, the simple statement vertex, and call vertex. Statements such as loops, conditionals, and assignments.
Parameter Vertex: This vertex is used where there is communication between two methods where a statement in one method calls the service of another method. It is passing a parameter between the two methods (caller and callee). There are four different type of this vertex which includes: the actual_in vertex, actual_out vertex, formal_in vertex and formal_out vertex. Linked to the method entry vertex.
Entry Vertex: this vertex represents the entry point of methods and classes. The method entry vertex signifies the entry point of a method while class entry vertex signifies a class entry point respectively.
Polymorphic Vertex**:** Represents the dynamic choice of possible bindings of objects in a polymorphic call.

JSDG Edges

An edge is a link/relationship that exists between statements of a program which is divided into different forms. Such as the control and data dependent edges that represent the transfer of control and movement of data among statements, parameter dependence edges, class and method membership edges, call edge and summary edge.
Control Dependence Edge**:** It is used to represents control dependence relations between two statement vertices.
Data Dependence Edge: It is used to represents data dependence relations between statement vertices.
Call Edge: connects a statement to a method through method entry vertex. Call edge also linked method call vertex a polymorphic choice vertex.
Parameter Dependence Edge**:** While passing parameters between a formal parameter and actual parameter through a call to a method, this edge can be used to represent the connection between the two parameters.
Summary Edge: when there is a transitive dependency between actual-n and actual-out, the summary edge is employed to represent this kind of dependency.
Class Member Edge: Represents the relationship between a class and methods the class. It connects a class entry vertex to all the methods entry vertices of that class.

3.2. Problem Formulation

In this section formulates the problem of test cases prioritization. Based on the following definitions, the propose approach to prioritize test cases was coined.

Cost-Cognizant Test Case Prioritization

Malishevsky et al [1] define cost-cognizant test case prioritization as follows:

Let T be test suite with n test cases with costs $t_1, t_2, \ldots t_n$ and F be a set of m faults revealed by T. Let $f_1, f_2, \ldots f_m$ be the severities of those faults.
Let TF_i be the first test cases to reveal faults i.

$$f = \frac{\sum_{i=1}^{m}\left(f_i \times \left(\sum_{j=TF_i}^{n} t_j - \frac{1}{2} t_{TF_i} \right) \right)}{\sum_{j=1}^{n} t_j \times \sum_{i=1}^{m} fi} \quad (1)$$

Dependency-based Cost-cognizant test award value:

The award value for the cost-cognizant faults dependence-based test cases prioritization was adopted from [17] when historical information record from the previous regression testing session is required to prioritize test case for the current regression testing process bearing the assumption the previous information remains

unchanged. The award for test casa revealing criticality of dependent faults is given by:

$$a_k = f_k \times \frac{df_{crit_{k-1}}}{\cos t_{k-1}} \quad (2)$$

Such that f_k stands for the total number of faults that revealed by a test case in the kth regression testing session $df_{crit_{k-1}}$ and $\cos t_{k-1}$ is the sum of the severities of dependent faults of the fault revealed by the test case and cost of a test case in the $k-1^{th}$ regression testing session respectively. Award value a_k is given by f_k multiplying to the result of dividing $df_{crit_{k-1}}$ by $\cos t_{k-1}$. A test case that has the highest value of the award function is ordered first.

To quantify how rapidly test suite detect dependencies among faults, evolutionary process (GA) with an objective function that computes the fitness of test case by computing the total severities of dependents faults to the total number cost incurred to reveal such faults times the number of functions exercised by the test case.

3.3. The ECRTP Technique

The ECRTP algorithm takes as input the source program, S, test suite, T, and produces its output as the prioritized test suite. The first step in the technique is the construction of JSDG model of the Java source code, M by using graphViz. Faulty version the original source code is then produced by mutating the original source code using µJava tool. The system then, identified the changed statements and updates the already constructed JSDG model, M'.

Algorithm: *ECRTP*
input: Java Source code P, Test suite T of P
output: *prioritizedT*
declaration:
begin
Construct JSDG model *M* of *P*
Create different faulty versions of *P*
Update the JSDG model *M*
Get all the affected statement
Get the test case coverage information through path-based testing *covInfo*
Identify and select the affected test cases
Encode the selected test cases
Generate initial population randomly
Evaluate the fitness of the initial population
Select two-best chromosomes for crossover
WHILE NOT TERMINATION
Perform crossover on the two selected chromosomes
Perform mutation the chromosomes formed after crossover
Evaluate the fitness of the newly formed chromosomes
Add two-best chromosomes to the population
return *prioritizedT*
end *ECRTP*

Fig. 1: ECRTP Algorithm

Path-based integration testing is performed on the updated JSDG to identify and get the coverage information of each test case. JSDG slicer identified the affected nodes from the changed statements as the slicing criteria on the updated JSDG model by identifying all the nodes that happen to be dependent on the changed statement. Later the algorithm identifies the dependencies that exist among the faults from the affected nodes. Test cases are then selected and later encoded to generate the initial population as a set of chromosomes. The fitness of each chromosome is then evaluated using the defined objective function.

The evaluation of the fitness of the chromosomes is performed by evaluating the fitness algorithm. The chromosomes serve as input to the algorithm and the output is the set of fitness score of each chromosome. For each gene in a chromosome, check whether an allele/node was executed by a gene/test case and identify and sum the severities of all the alleles (nodes) that are dependent on the executed allele (node). We used a random number to assign severities to a fault by considering the important of a component to the system as was applied in [38, 39].

After evaluating the fitness of each chromosome, ECRTP algorithm selects two best chromosomes from the population for crossover according to their fitness score, performs crossover on the two selected chromosomes and then mutate the offspring before evaluating the fitness of the offspring using the same process as that of their parents. The two offspring are added to the pool and the remaining are ignored from the new population until a maximum number of iterations is reached.

3.4. Experiment

Consider a test suite, T, containing five test cases t1, t2, t3, t4, t5 and the faults exposed by each test case be F (f1, f2, f3, f4, f5) as shown in Table 1. Table 2 represents the dependencies that exist within the faults being exposed by the test cases.

ECRTP algorithm checks all the faults from the affected nodes if there are any faults occur as result of the occurrence of another fault, then that fault is marked as a fault that is dependent on that fault that leads to its occurrence.

Table 1: Test cases and fault exposed

	Faults				
Test case	**F1**	**F2**	**F3**	**F4**	**F5**
t1		×			
t2		×	×		×
t3	×		×		
t4	×		×		
t5		×	×	×	

The algorithm presented in Figure 1. then stores the dependency information as indicated in Table 2.

Table 2: Faults dependence matrix

↓→	**F1**	**F2**	**F3**	**F4**	**F5**
F1	0	0	0	0	0
F2	1	0	0	0	0
F3	0	0	0	0	0
F4	1	1	0	0	0
F5	1	0	1	0	0

The algorithm then uses the information from Table 1. and Table 2. It also considers the cost of running each test and the severity of each fault revealed. Table 3 and Table 4 give an example of test cases and their costs and faults and their severities.

Table 3: Example of test case and their costs

Test case	**Cost**
1	2
2	1
3	3
4	5
5	4

Table 4: Example of faults and their severities

Fault	**Severity**
1	4
2	3
3	5
4	2
5	1

Test cases are then encoding it into integer numbers using permutation encoding such that genetic algorithm will be able to operate on it. After test case encoding of the selected test cases, the algorithm goes further to generate the initial population randomly. This serves as the first generation of the population which undergoes crossover and mutation to produce the next generation. For any chromosome to move to next generation after it has evolved, the fitness value calculated by the formula shown in (1) of that

chromosome has to be evaluated, chromosome's fitness is then compared. The chromosome that survives to the next generation are the two chromosomes with the highest fitness value. These two selected chromosomes undergo the genetic process of recombination to produce another generation from which best offspring are selected and compared. The process continues until the termination condition is reached.

4. Conclusion

This research proposes an evolutionary regression test cases prioritization for object-oriented programs based on the dependency among faults. The technique award value to test case based the severities of the dependent faults the test case was able to detect and the cost the test case to reveal such severities. Test cases are encoded to form chromosome which is described as a population by the genetic algorithm.

Acknowledgement

We acknowledge the Ministry of Higher Education, Malaysia for supporting this research work with fund under the Fundamental Research Grant Scheme FRGS/1/2015/ICT01/UPM/02/12 awarded to the Faculty of Computer Science and Information Technology, University Putra Malaysia.

References

[1] Yoo S & Harman M, (2007), "Regression Testing Minimisation, Selection and Prioritisation: A Survey," *Softw. Testing, Verif. Reliab.*, vol. 7, pp. 1–30.

[2] Catal C & Mishra D, (2013), "Test case prioritization: A systematic mapping study," *Softw. Qual. J.*, vol. 21, no. 3, pp. 445–478.

[3] Singh Y, Kaur A, Suri B & Singhal S, (2012), "Systematic Literature Review on Regression Test Prioritization Techniques," *Informatica*, vol. 36, pp. 379–408.

[4] Shrivathsan AD, Ravichandran KS & Sekar KR, (2015), "Test suite reduction mechanisms: A survey," *Int. J. Appl. Eng. Res.*, vol. 10, no. 18, pp. 39841–39848.

[5] Mohapatra SK & Prasad S, (2015), "Test Case Reduction Using Ant Colony Optimization for Object Oriented Program," vol. 5, no. 6, pp. 1424–1432.

[6] Khan SUR, Lee SP, Parizi RM & Elahi M, (2014), "A code coverage-based test suite reduction and prioritization framework," *2014 4th World Congr. Inf. Commun. Technol. WICT 2014*, pp. 229–234.

[7] Beena R & Sarala S, (2013), "Code Coverage Based Test Case Selection," *Int. J. Softw. Eng. Appl. (IJSEA), Vol.4, No.6, Novemb. 2013*, vol. 4, no. 6, pp. 39–49.

[8] Wang S, Ali S, Gotlieb A & Liaaen M, (2016), "A systematic test case selection methodology for product lines: results and insights from an industrial case study," *Empir. Softw. Eng.*, vol. 21, no. 4, pp. 1586–1622.

[9] Wang Y, Zhao X & Ding X, (2015), "An Effective Test Case Prioritization Method Based on Fault Severity," in *Software Engineering and Service Science (ICSESS), 2015 6th IEEE International Conference*, pp. 737–741.

[10] Malishevsky G, Ruthruff JR, Rothermel G & Elbaum S, (2006), "Cost-cognizant Test Case Prioritization," Technical Report TR-UNL-CSE-2006-0004, Department of Computer Science and Engineering, University of Nebraska–Lincoln, Lincoln, Nebraska, U.S.A., 12 March 2006.

[11] Rava M & Wan-Kadir WMN, (2016), "A Review on Automated Regression Testing," *Int. J. Softw. Eng. Its Appl.*, vol. 10, no. 1, pp. 221–232.

[12] Beszedes, T. Gergely, L. Schrettner, Jasz J, Lango L & Gyimothy T, (2012), "Code coverage-based regression test selection and prioritization in WebKit," *IEEE Int. Conf. Softw. Maintenance, ICSM*, pp. 46–55.

[13] Musa S, Sultan AB, Abdl Ghani AA, & Baharom S, ,(2014)"A Regression Test Case Selection and Prioritization for Object- Oriented Programs using Dependency Graph and Genetic Algorithm," *Int. J. Eng. Sci.*, vol. 4, no. 7, pp. 54–64.

[14] Elbaum S, Kallakuri P, Malishevsky A, Rothermel G & Kanduri S, (2003), "Understanding the effects of changes on the cost-effectiveness of regression testing techniques," *Softw. Testing, Verif. Reliab.*, vol. 13, no. 2, pp. 65–83.

[15] Elbaum S, Malishevsky AG & Rothermel G, (2002), "Test case prioritization: a family of empirical studies," *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, pp. 159–182.

[16] Jovanovic S, (2009), "Software Testing Methods and Techniques," *IPSI BgD Trans. Internet Res.*, vol. 5, no. 1, pp. 30–41.

[17] Hao D, Zhang L, Zang L, Wang Y, Wu X, Xie T & Member S, (2016), "To Be Optimal or Not in Test-Case Prioritization," vol. 42, no. 5, pp. 490–504.

[18] Goel N & Sharma M, (2015), "Prioritization of Test Cases and Its Techniques," *Int. J. Comput. Appl.*, vol. 5, no. 4, pp. 85–89.

[19] Lou Y, Hao D & Zhang L, (2015), "Mutation-based test-case prioritization in software evolution," *2015 IEEE 26th Int. Symp. Softw. Reliab. Eng. ISSRE 2015*, pp. 46–57.

[20] Di Nardo D, Alshahwan N, Briand L & Labiche Y, (2015), "Coverage-based regression test case selection, minimization and prioritization: a case study on an industrial system," *Softw. Testing, Verif. Reliab.*, pp. 371–396.

[21] Kumar H & Chauhan N, (2015), "A Module Coupling Slice Based Test Case Prioritization Technique," *Int. J. Mod. Educ. Comput. Sci.*, vol. 7, no. 7, pp. 8–16.

[22] Kayes I & Chakareski J, (2014), "The Network of Faults: A Complex Network Approach to Prioritize Test Cases for Regression Testing," *Innov. Syst. Softw. Eng.*, vol. cs.SE, pp. 1–13.

[23] Wang Y, Zhao X, & Ding X, (2015), "An effective test case prioritization method based on fault severity," in *Proceedings of the IEEE International Conference on Software Engineering and Service Sciences, ICSESS*, 2015, vol. 2015–Novem, pp. 737–741.

[24] Ranga K, (2015), "Analysis and Design of Test Case Prioritization Technique for Regression Testing," *IJIRST –International J. Innov. Res. Sci. Technol.*, vol. 2, no. 01, pp. 248–252.

[25] Di Nucci D, Panichella A, Zaidman A, & De Lucia A, (2015), "Hypervolume-Based Search for Test Case Prioritization," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 9275, pp. 157–172.

[26] Silva A, (2012), "A vocabulary of program slicing-based techniques," *ACM Comput. Surv.*, vol. 44, no. 3, pp. 1–41.

[27] Larsen L & Harrold MJ, (1996), "Slicing object-oriented software," *18th Int. Conf. Softw. Eng.*, pp. 495–505.

[28] Horwitz S, Reps T, & Binkley D, (1988), "Interprocedural slicing using dependence graphs," *Proc. ACM SIGPLAN 1988 Conf. Program. Lang. Des. Implement. - PLDI '88*, vol. 12, no. 1, pp. 35–46.

[29] Panigrahi CR & Mall R, (2013), "An approach to prioritize the regression test cases of object-oriented programs," vol. 1, no. June, pp. 159–173.

[30] Walkinshaw N, Roper M, & Wood M, (2003), "The Java system dependence graph," *Proc. - 3rd IEEE Int. Work. Source Code Anal. Manip. SCAM 2003*, pp. 55–64.

[31] Donglin L & Harrold MJ, (1998), "Slicing objects using system dependence graphs," *Softw. Maintenance, 1998. Proceedings. Int. Conf.*, no. November, pp. 358–367.

[32] Sinha S, Harrold MJ, & Rothermel G, (1999), "System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow," *Softw. Eng. 1999. Proc. 1999 Int. Conf.*, no. May 1999, pp. 432–441.

[33] Meng Y, Xu D, Zhang Z, & Li W, (2015), "System Dependency Graph Construction Algorithm Based on Equivalent Substitution," *2015 Eighth Int. Conf. Internet Comput. Sci. Eng.*, pp. 106–110.

[34] Harman M & Danicic S, (1995), "Using program slicing to simplify testing," *Softw. Testing, Verif. Reliab.*, vol. 5, no. 3, pp. 143–162.

[35] Mohapatra DP, Mall R, & Kumar R, (2006), "An overview of slicing techniques for object-oriented programs," *Inform.*, vol. 30, no. 2, pp. 253–277, 2006.

[36] Kovács G, Magyar F, & Gyimóthy T, (1996), "Static slicing of java programs," 1996.

[37] Zhao X, (1998), "Applying Program Dependence Analysis To Java Software," *Work. Softw. Eng. Database Syst. Int. Comput. Symp.*, no. June 2001, pp. 162–169, 1998.

[38] Elbaum S, Malishevsky A, & Rothermel G, (2001), "Incorporating varying test costs and fault severities into test case prioritization," in *Proceedings - International Conference on Software Engineering*, 2001, pp. 329–338.

[39] Kavitha R & Sureshkumar N, (2010), "Test Case Prioritization for Regression Testing based on Severity of Fault," *Int. J. Comput. Sci. Eng.*, vol. 02, no. 05, pp. 1462–1466.