Microsystem Technologies (2020) 26:1463–1477
https://doi.org/10.1007/s00542-019-04679-7

TECHNICAL PAPER

Regression test case selection and prioritization for object oriented software

Dharmveer Kumar Yadav[1] · Sandip Dutta[1]

Received: 19 October 2019 / Accepted: 25 October 2019 / Published online: 6 November 2019

Abstract
In software maintenance, after modifying the software a system needs regression testing. Execution of regression testing confirms that any modified code has no adverse effect as well as does not introduce new faults in the existing functionality of the software. When working with object-oriented programming code-based testing is generally expensive. In this study, we proposed a technique for regression testing using unified modeling language (UML) diagrams and code-based analysis for object-oriented software. In this research work, the design and code based technique with an evolutionary approach are presented to select the best possible test cases from the test suite. We used the dependency graph for intermediate representation for the objectoriented program to identify the change. The selection of test cases is done at the design level using the UML model. The models are compared to identify the change between these two models. The proposed approached maximizes the value of APFD.

1 Introduction

When changes are performed to software then regression testing is done. Regression testing is done to ensures that existing software does not change the existing functionality software when codes are changed. In the maintenance phase, the software may be enhanced the existing software. Regression testing confirms that no new error is introduced while code is modified (Malishevsky et al. 2006). When software modification is made if we rerun all the test cases it takes more time. Thus regression testing activity time consuming so tester needs to save effort and cost to run the cases. In regression testing, we run some of the tests from the suite (Rothermel and Harrold 1997). So selects a subset of tests from existing suite testing cost may be minimized. In prioritization, test cases are scheduled to achieve some goals such as code coverage, fault detection. The various method has been presented for regression testing. Regression testing includes minimization of the test suite, selection of tests and prioritization method. The process minimization test suite is to identify duplicate test cases and then eliminate those test cases. The selection of tests is a process to select a test to test the modified program. Finally, prioritization of test cases includes ordering of tests to meet some objectives such as fault can be detected early, coverage of source code, etc.

The software needs continuous development during the software lifecycle for several reasons such as add new features, changing business requirements, due to change in technologies, improve quality, correcting faults, etc. When software is modified then it is important to confirm that software is not badly affected by changing the software. Software maintenance is an important phase of the overall lifecycle. Among the several maintenance works, regression testing plays an important role. A regression test is performed during the maintenance phase. The retest-all method consists rerunning of all test cases from the test suite, which is very costly, not efficient and undesirable in the maintenance life cycle (Rothermel and Harrold 1996). An alternative method, called selective retest method which considers that all parts are not affected by the modification. The selection method for regression testing selects a suitable subset of tests and runs the test case from the existing suite to validate the behavior of changed software. Regression testing makes confidence in the developer that modified code does not have a bad impact on other parts of code (Graves et al. 2001; Engstrom et al.

✉ Dharmveer Kumar Yadav
kumar.dharmveer@gmail.com

Sandip Dutta
sandipdutta@bitmesra.ac.in

[1] Department of Computer Science and Engineering, Birla Institute of Technology, Mesra, Ranchi, JH, India

2008). This reduces the regression testing cost. If a selective approach exposes the identical faults as exposed by retest all method then it is called safe. Regression testing needs to handle various issues like how to identify where a modification has occurred and which parts are impacted due to modifications, test selection needs to select those test which detects more faults in software after modifications, running the test case to verify the behavior of the software. Various researchers in literature have proposed regression testing based on a code-based approach. Many researchers proposed a different method for testing and study different types of code-based approach, however, it has certain flaws: it is costly when software systems are complex and large. The code-based method considered to be good for the unit test but it has scalability. Therefore, it is not suitable for large components (Skoglund and Runeson 2005). Sometimes it becomes hard to recognize changes in code. So the code-based approach is dependent on the programming language. Regression testing models are also used for object-oriented programs (Engstrom et al. 2008). The model-based approach is independent of any programming language. The model-based approach gains many benefits than code-based testing. But this approach has some limits (Fahad and Nadeem 2008).

The proposed method is the firewall technique, slicing method, and control graph of the program (White et al. 2005; Iqbal et al. 2007). UML technique used to categorize the tests after modification done in the code into retestable, reusable, and obsolete (Briand et al. 2006). They compared UML models of class and sequence diagram for modification. To identify the change selective test method used with class and state diagrams (Iqbal et al. 2007). UML sequence and UML class diagrams are used for test selection Pilskalns et al. (2006) proposes a UML-based regression testing for software (Batra 2009). They derive a sequence diagram of component then it is converted to a control flow graph. To detect the change graph is traversed which is traversed. The software needs to change with change in user requirements to meet the user's requirements. Analysis of Change impact is a method to determine effected code in software (Maia et al. 2010). It has a great impact on the maintenance of software and regression testing (Rovegard et al. 2008). Many researchers worked on change impact analysis. Some approaches are traceability based examination others are based on dependency relationships to know the effects of change. Traceability analysis is based on requirements, designs, source code and tests (Badri et al. 2005). Dependency based method is an analysis of syntax (Law and Rothermel 2003) Most of the techniques use a source code based method for change impact and very few works on requirements and designs (Hewitt and Rilling 2006; Shiri et al. 2007; Briand et al. 2006).

The paper is organized as follows: Sect. 2 consists of related work used in the research work. In Sect. 3, background concepts related to the proposed work is presented. In Sect. 4, introduction to Genetic algorithm (GA) is shown. Section 5 presents proposed work with specific steps. In Sect. 6 result and comparison with other technique is shown. Finally, the conclusion is shown in Sect. 7.

2 Related work

Several researchers have presented prioritization of test cases like Greedy method, Optimal algorithms technique non- evolutionary Logarithmic least-square Weight least square method evolutionary algorithms and fault detection (Rothermel and Harrold 1997; Al-Salami 2009; Byson 1995; Chu et al. 1979; Askarunisa et al 2010). Prioritization using the genetic algorithm and historic information was proposed (Huang et al. 2010). The authors proposed a prioritization technique using scenarios with the help of a genetic algorithm (Sabharwal et al. 2011). The author presented work using a genetic algorithm to find the most suitable test case. In 2009, an algorithm was proposed using a Genetic algorithm for branch coverage (Rothermel and Harrold 1997). The new search algorithm is proposed for prioritization (Li et al. 2007). For C++ and Java programs graphs and class control flow graph is proposed (Rothermel et al. 2000; Harrold et al. 2001). This method finds differences between old versions and a new version of the Java class dependency graph of class (Le Traon et al. 2000). Their technique decomposes the graph into connected components. If the component is changed, then again all the connected components are tested. The activity diagram describes the requirements of the system and builds a graph for the class diagram using class specification and information of implementation (Kundu and Samanta 2009). The researcher performed regression testing using class, state machine using a UML diagram (Iqbal et al. 2007). Proposed test case prioritization using a genetic algorithm (Yadav and Dutta 2016). The fault-based method is presented for prioritization using fuzzy logic (Yadav and Dutta 2017). Many researchers proposed regression testing based on dataflow analysis (Rothermel and Harrold 1994b; Harrold and Soffa 1989). This approach detects modified pairs for variables used in the program and selected test cases are executed. Types of a variable used are divided into computation uses variable and predicate uses variable. The computation variable is used for computations, and predicate use variables used for the conditional statements. The dataflow coverage-based method is proposed to detect changes across multiple procedures (Agrawal and Arvinder 2018). The slicing-based approach was suggested, which uses inter-procedural

for slicing (Gupta et al. 1996). This approach uses a backward and forward slicing approach to identify change. The regression selection technique uses a program slicing approach (Panda et al. 2016). In this technique when a test case executed on the modified program then selection of test is used for regression testing. They do not use any intermediate representation of programs. Regression selection method using traversing the control flow graph of programs. This approach produces a better result as compared to the graph walk-based technique. Test case prioritization proposed using total and additional methods (Hao et al. 2014). A similarity concept is used for the prioritization method is proposed (Fang et al. 2014). In this work test case, the execution profile was utilized. Multi objective-based prioritization is suggested for regression testing (Marchetto et al. 2016). Prioritization using source code and dynamic based method may be a more useful technique (Luo et al. 2016). The authors proposed a method using the various case study of an industrial system using regression faults (Di Nardo et al. 2015). Very few prioritizations focus on on-suite change (Lu et al. 2016). The optimization technique based on the ant colony technique is proposed for the prioritization of test cases (Agrawal and Arvinder 2018; Singh et al. 2010). The statement level test selection approach is used for object-oriented programs (Musa et al. 2015). Prioritization of test case is performed based on fault detection and clustering approach for object oriented software (Yadav and Dutta 2019a, b).

3 Background concepts

We discuss some important models that are used in our research work. We first explain all readily established models used for object-oriented programming. We explain some concepts and definitions used for regression testing.

3.1 Graph representation for object-oriented programming

Some important features of the object-oriented program is an inheritance, polymorphism, encapsulation, etc. To represent relationship among program elements, several graph model is proposed by researchers for object-oriented program (Biswas et al. 2011) like call graph (Rothermel and Harrold 1997), Interprocedural program dependency graph (Rothermel and Harrold 1994), class dependency graph (Rothermel and Harrold 1994), Java inter-class graph (Briand et al. 2006), and dependency graph for object (Najumudheen et al. 2009). Here, we discuss some of the dependency graph models for the C++ program and Java. The dependency graph is a collection of a set of vertices as V and set of edges as E. System Dependency Graph is extended (ESDG) as a model used for the intermediate representation of object-oriented program, earlier it was used (Horwitz et al. 1990; Larsen and Harrold 1996; Panigrahi and Mall 2012). ESDG model represents control dependency and data dependencies of the program. This graph model helps to identify changes done in programming. ESDG can be defined as directed graph G = (V, E), which consists set of vertices and edges (V, E) in graph G.

3.1.1 ESDG types of vertices

The method or Statement vertex: Statement vertex V is used to represent the statement which is present in the method body.

Entry vertex Class and method have entry vertex V. Entry vertex represents the entry in class and method.

Parameter vertex Parameter vertex V represents how to pass the parameter between a caller and callee method. This is divided into four types: Formal-in, Formal-out, Actual-in and Actual-out. For each call vertex Actual-in and Actual-out vertex are made and for all method entry vertex Formal-in and Formal-out vertex is formed.

3.1.2 ESDG types of edges

Control dependency edge Control dependency relation between two statement vertex is represented by control dependency edge.

Data dependency edge Data dependence relation between statement vertex is represented by data dependency edge.

Call edge This connects a calling statement to the method entry vertex.

Parameter dependency edge Parameter dependency edge used when values are passed between actual and formal parameters in a method call.

Class member edge Represents relation between a class and class methods is represented by the class member edge.

Inheritance edge Inheritance relationships between classes can be represented by an inheritance edge. In Fig. 1, example 1 shows a graphical representation of the C++ Program.

Example 1: Graph representation of C++ Program:

```
Entry Vertex: class Add {
S2: int m;
S3: int n;
E4: public void read(int i, int j) {
S5: m =i;
S6: n=j;
}
E7: Public int sum()
{
S8: int s=m+n;
S9: return s; }
};
```

3.2 UML model

UML is used as a design model for object-oriented systems. UML has an important role in this object-oriented analysis and design. UML is divided into structural and Behavioral diagrams. The structural diagram captures the static part of the system like class diagram, object diagram, etc. Behavioral diagrams are also called dynamic diagrams such as sequence diagrams, collaboration diagrams, etc. The sequence diagram represents how processes interact with each other. Interactions of an object shown through a sequence diagram using a time sequence. A sequence diagram represents parallel vertical lines is also called lifeline, different objects that live simultaneously, messages exchanged between the objects. The test case is generated with the help of the sequence diagram. We have considered the case study of an ATM to generate a test case. We have drawn the UML sequence diagram and use case diagram shown in Figs. 2 and 3 for ATM.

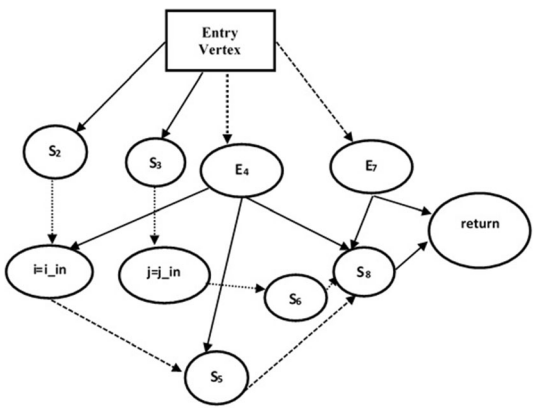


Fig. 1 System dependency graph for C++ program

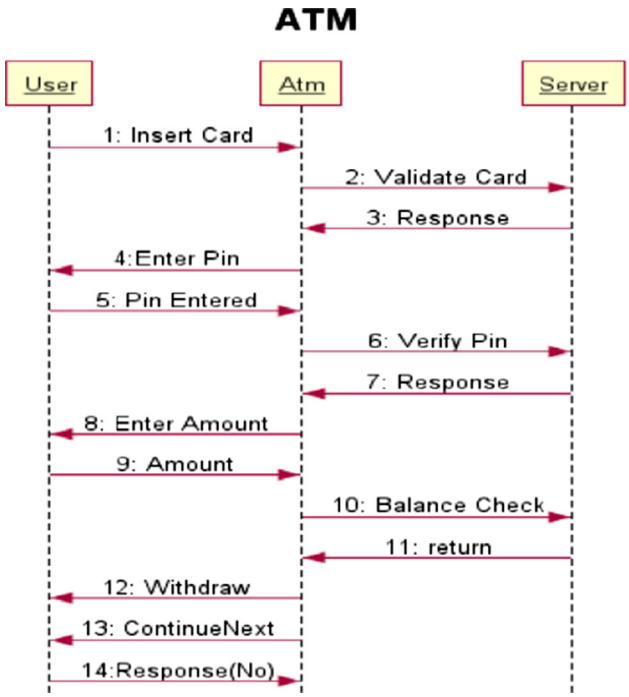


Fig. 2 Sequence diagram for ATM

4 Genetic algorithm

In 1975, a new idea called Genetic algorithms (GAs) in "Adaptation in natural and artificial systems" (Holland 1975). GA is a heuristic technique based on "Survival of the fittest" and it solves search and optimization problems. GA technique is applied to search for the best solution in search space and finds an approximate solution to optimization and search problem. GA has a population of all possible solutions. Chromosome represents each solution which is an abstract representation. In GA first solutions are coded into chromosomes. Reproduction operators are applied to chromosomes directly, mutations are performed then recombination is performed.

GA has following steps:

SELECTION: In the selection process individuals are selected randomly for reproduction with a probability depends on fitness of individuals.

REPRODUCTION: In this step, offspring are generated. To generate new chromosomes, GA use recombination and mutation.

EVALUATION: In this step fitness is evaluated for new chromosomes.

REPLACEMENT: In replacement individuals are mutated from old population and replaced when optimal solution is found then algorithm is stopped.

Encoding Chromosome: The chromosome is encoded in a binary string, it looks like this:

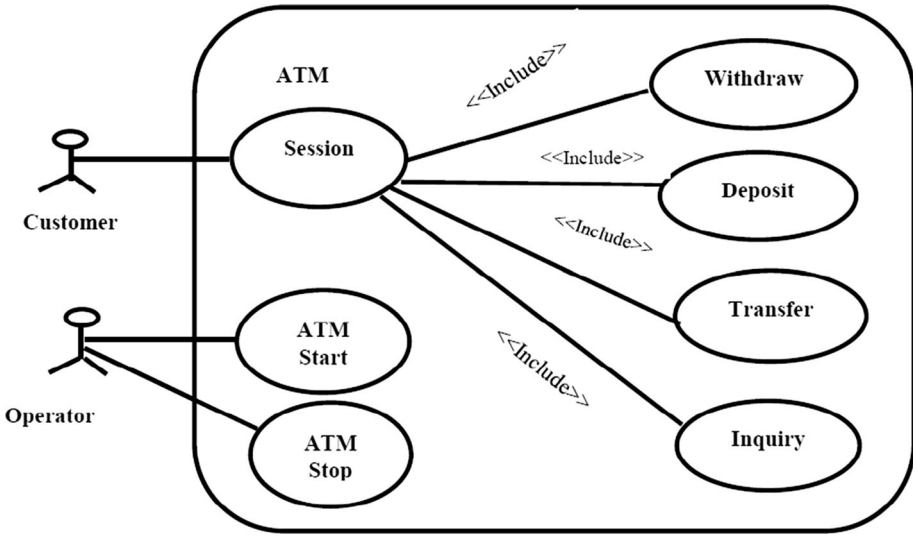


Fig. 3 Use case diagram of ATM

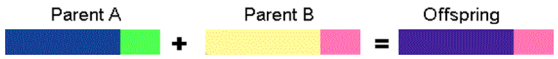
Chromosome1	1011100100111010
Chromosome2	1001111000011010

There are a number of ways in which a chromosome can be encoded. This encoding technique generally depends on the problem solution.

Crossover and mutation: GA has two important operators one is a crossover and the second is a mutation. GA performance dependents on these two operators. In this paper, we have taken some examples for several encoding methods like binary encoding, Permutation, Value-based encoding, and Tree-based encoding. Generally, 80–90% is the high crossover rate. The mutation rate must be the low value (0.5–1%).

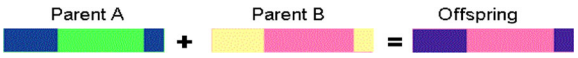
Crossover for binary encoding:

(a) Single point crossover—First single point is selected then a string of parent A is copied to the crossover point, rest is copied from parent B.



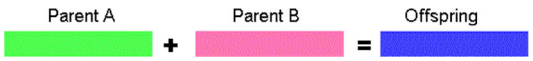
01101011+11011110 = 01101110

(b) Two-point crossover: Offspring produced from start to the first crossover point is copied from the parent then remaining copied part from the first to the second crossover point from the second parent and rest from the first parent is copied.



00110100 + 010**110**01=00110000

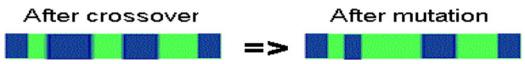
Arithmetic crossover—New offspring are created using some arithmetic operation.



10101111 + 10110111 = 10100111 (AND)

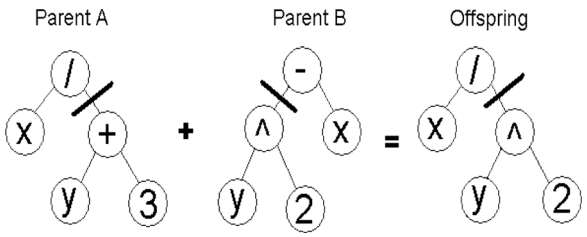
Uniform crossover—Randomly bits are copied from the first or second parent.

The mutation for binary encoding: invert selected bit.



Crossover for tree encoding:

Tree crossover—One crossover point in both parent are selected, parents are separated in that point to produce offspring parts below crossover point is exchanged.



5 Proposed methodology

The proposed approach consists selection of test cases using source code analysis with UML Model and test case prioritization using a genetic algorithm (GA). The proposed steps are as follows:

Step 1: Identify the change impact analysis
Step 2: Regression test case selection
Step 3: Test case prioritization using GA
Step 4: Performance analysis

5.1 Identify the change impact analysis

The modified software system may have an adverse effect or ripple effect (Bohner 1996). Modified software need to identify the impact of change in code. Change impact of software identify side effects due to change in a program like the error is introduced in the program. Ripple effect measures effect to other parts of the module due to changes made to a program (Black 2008; Mansour and Salem 2006). Change impact analyzes the request for change and identifies change which consists set of affected possible elements by making a change request. Then impact analysis method, estimate change set identify what are the different module of software which is affected by the modification.

We used the automated teller machine (ATM) in this paper as a case study. We show the procedure followed by the case studies. In Fig. 4 procedure is shown using the activity diagram. First, we define a system of different types of logical changes to a system. Three types of logical changes are used: change in requirement, design

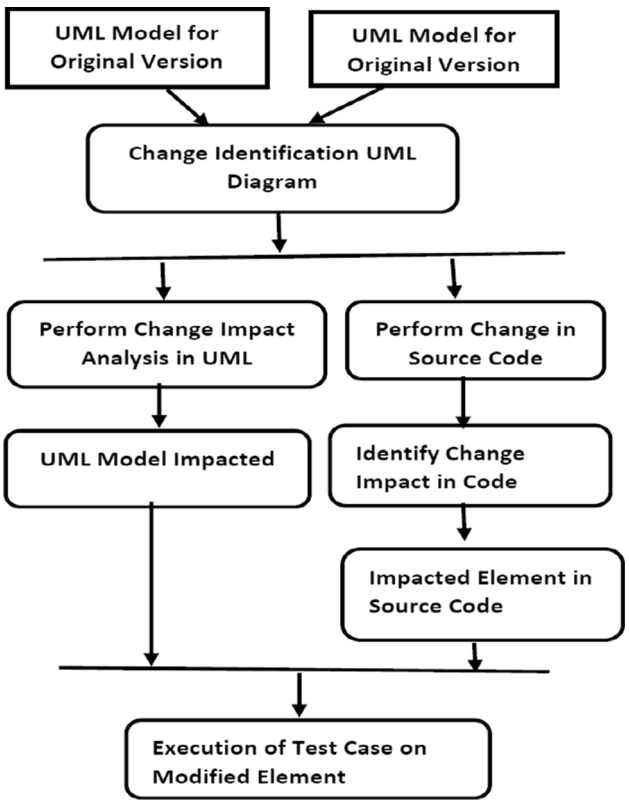


Fig. 4 Process used for change identification

improvements, and correcting the error. We then developed a tool to analyze the changes between the original UML diagram and the modified UML diagram. This gives a set which contains impacted elements in the UML diagram. Concurrently, program source code is also examined. From the source code, the direct and indirect impact is analyzed from models. First, logical changes are performed then identify the set of elements changed in UML and source code. Then perform testing for revised code to make sure that added or modified functionalities of the software were implemented correctly and there is no impact on unmodified functionalities. Changes are recognized by matching the original and changed version of the code. Impacted elements are identified and grouped into a set. Then results are verified. The stepwise process is shown in Fig. 4 for change identification.

5.2 Regression test case selection

This research work presents a methodology for RTS and regression test case prioritization(RTP) for object-oriented programs. The test case is the prioritization approach prioritize those tests first which have a high capability of detecting the fault rate. Different types of change possible in the program are shown in Fig. 5.

5.2.1 Types of changes

The main problem in regression testing is to determine changes occurred between original program P and modified program P'. Once the change is recognized, test cases are selected to run only related to changes in the program. In this work RTS approach use, the ESDG dependency graph model is proposed for intermediate representation of the original program and modified the program to analyze the changes in code. RTS approach first uses a depth-first search (DFS) for traversing the ESDG graph of two program versions to identify the set of edges, which may cause the behavior of the program to make a change in the modified version of the program. Then, for each test from the test suite, the method checks its coverage information of the old version of the program with the set of identified edges traversed from the graph. Semantic analysis identify changes in the original program and modified the program. The mutation operators are suggested to detect possible faults in object-oriented programming. It is a white-box testing method that measures the efficiency of test cases. Initially, faults are introduced in the original program by mutant versions. Mutants are created by mutation operators in the program which produces syntactic changes in the programs. Then test cases will be executed on original and mutant programs so that it will discover faults in the program. As shown in Fig. 5a if superclass A is added then

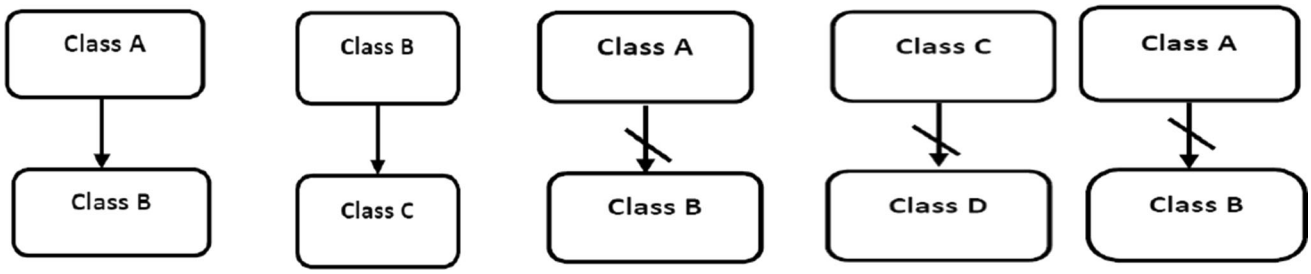


Fig. 5 **a** Add superclass A. **b** Add subclass C. **c** Delete super class A. **d** Delete subclass D. **e** Delete edge

subclass B must be modified or affected Fig. 5b when subclass C is newly added then or inherited from class B then there is no change in class B. Figure 5c when superclass A is deleted then all subclass which is inherited from superclass will be affected, so class B will be affected. Figure 5d when subclass D is deleted then class C is affected Fig. 5e if an edge is removed then no effect on class A.

5.3 Test case prioritization using GA

One of the population-based searching techniques is the Genetic algorithm. GA technique applied to solve various problems to find the best solution. GA was first introduced by (Holland 1973). GA tries to find the best solution using a defined fitness function. Steps of GA is shown in Fig. 6.

Algorithm Test selection

Input: T-Test Case

T'- Tests selected from T to retest MSD

OSD- Original sequence diagram

MM- Modified Method

MSD- Modified sequence diagram

TS- Test Suite

Output: T', TS

begin

T'={ }

MM'= identify changed method(OSD, MSD)

Select Unit test for affected sequence diagram

(i)First select test cases set T to execute for OSD

(ii) Then select $T \subseteq T'$ to retest MSD' with T' to find the accuracy of modified program P' w.r.t. T'.

(iii) Create T" if necessary, new test case set for modified program MSD' for retest P' with test case T", so that same correctness of MSD'is obtained with respect to T''.

(iv) create another T"' from T, T', T" by adding new test cases, to test MSD correctness.

Return T' and TS

End

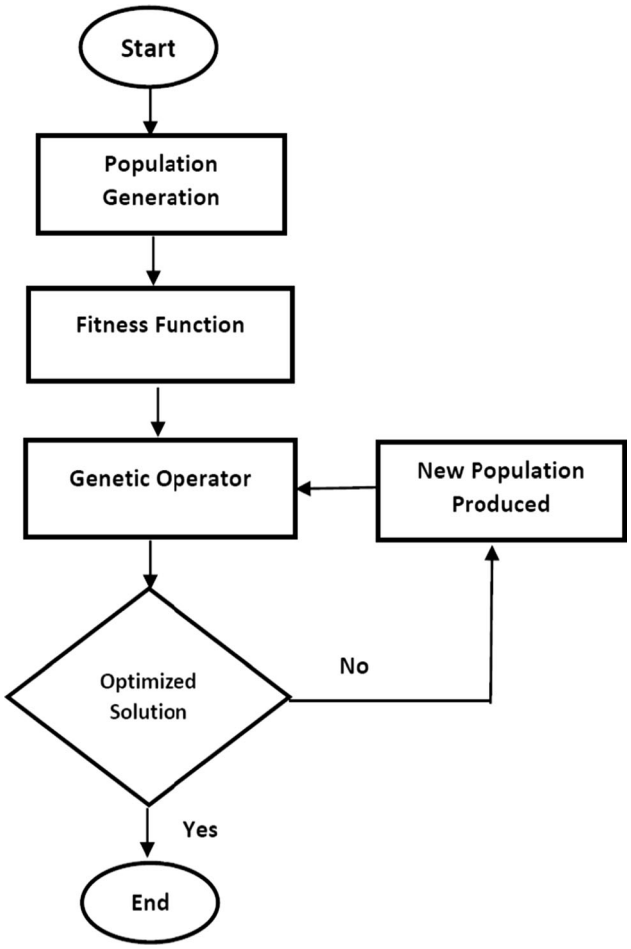


Fig. 6 Flow chart for the genetic algorithm

Step 1—Start: Initialize generated population, suppose N is the numbers of chromosomes {c1, c2… cn}, then Test Suite = Total no. of chromosomes (N).

Step 2—Fitness: Calculation of fitness function f(x)

Step 3—New population: By repeatedly following steps.

Step 4—Selection: Select the best suitable chromosome from the population-based on a Fitness function.

Step 5—Crossover: Operator is applied

```
For selected chromosomes
        While (all faults are detected)
         Perform crossover
         Perform mutation
         Duplicate is removed
           end while
End for
```

5.3.1 Generate test case

Initially, the population is generated randomly. The solution to a given problem is represented by each chromosome in the population. Chromosome denotes a set of values for the input variable.

Test Suite = Total no. of chromosomes (N)

Estimate Fitness Function:

The fitness function selects the test cases which have the maximum number of statement coverage and maximum fault detected by the test cases. An objective function determines whether chromosomes are good or bad. This function is a real number which is used to compare two or more test cases.

In these paper objectives function is considered to find the fitness of test cases. Fitness function is shown in Eq. (1 and 2). Objectives function are statement coverage and fault coverage which is discussed below:

(a) Statement coverage: Statement coverage can be defined as number of statement covered in a given program by a n number of test cases such as $\{T_1,T_2,T_3,\ldots T_N\}$. Assume there is m no. of statement in program P which is $\{S_1, S_2, S_3, S_4,\ldots S_m\}$. Consider statement coverage ST is a function which return the number of statement S covered by T_i.

$$ST_i = \sum S(T_i)/m \tag{1}$$

m is the total no. of statement in program and n is the number of test cases.

(b) The objective function is fault coverage which is explained below:

Fault Detection Rate (FDR) can be defined as:
FDR=(No. of Faults Detected/Total No. of Faults).

$$\text{Fitness Function} = \text{Max}\ \{(\text{FDR})\}\ \text{and}\ \text{Max}\{ST_i\} \tag{2}$$

$F_1(X_i)$ = Max(ST_i)
$F_2(X_i)$ = Max(FDR)
$F(X_i)$ = max { F1(X_i) } and{ F2(X_i) }

5.3.2 Apply crossover and mutation

In GA—crossover probability and mutation probability are two important parameters in GA. If the crossover is performed from some parts of the parent's chromosome, offspring are produced. If the value of cross probability is a hundred percent, it means that offspring are created by crossover. If the value of cross probability is 0%, it means that the entire new generation is ended from the old population and has the same copy of a chromosome. Crossover is performed in such a way that new chromosomes contain good things from old chromosomes so that a better chromosome will be produced. In mutation, probability mutation is performed after crossover when there is no change in offspring. If the probability of mutation is 100%, then

Table 1 Mutation operators

Class	Operator	Example
Inheritance	IOD	Overriding method deletion
	IOP	Overriding method calling position change
	IOR	Overriding method rename
	ISK	Super keyword deletion
Overloading	OMR	Overloading method contents change
	OMD	Overloading method deletion
	OAO	Argument order change
	OAN	Argument number change
Encapsulation	AMC	Access modifier change
Java specific features	JSC	static modifier change
	JTD	this keyword deletion
	JDC	Java-supported default constructor creation
	JID	Member variable initialization deletion
Polymorphism	PMD	Instance variable declaration with parent class type
	PNC	new method call with child class type
	PPD	Parameter variable declaration with child class type
Programming mistakes	EAM	Accessor method change
	EMM	Modifier method change

the entire chromosome is changed, if the probability is 0% then no change. The mutation is not performed always, because GA will fall into a random search. GA can minimize time to run remaining test cases if the duplicate test case can be deleted. The algorithm can be terminated if an optimized result is obtained. When there is no change in fitness function then the algorithm can be stopped. The initial size of the test case is 50, a single-point crossover is selected, the initial crossover probability is 0.85 range is 0.1 to 1.0 and the initial mutation probability is 0.25 parameters are used. Mutation operators applied to detect faults for object-oriented programs (Kim et al. 2000). This set of 18 mutation operators are listed in Table 1.

5.4 Performance analysis

To calculate the performance of proposed work we used fault exposing potential and APFD metric.

5.4.1 Total fault-exposing-potential (FEP)

The capability of test cases to detect faults is called fault exposing potential (FEP). This depends on the ability to detect faults by running the faulty lines and also depends on the probability that a faulty statement makes a failed test case (Rothermel et al. 2000). So, to achieve fault-exposing-potential we apply mutation analysis of a test case. Mutation analysis generates "mutants" of java program by changing the program statements. This also evaluates the quality of testing by assessing whether those faults can be detected by test case i.e killing those mutants. FEP can be defined as follows:

Definition Program P, test suite T is given, we create set N of m mutants i.e. N = {n_1, n_2, n_3, n_4, n_m } of P. Knows which statement S_j in program P holds each mutant. For each test case t_i belongs to T, we perform to run each mutant variety n_k of program P on t_i. Observing whether ti kills that mutant. Having this information about for each test and mutant, considering each of t_i and S_j in P and obtain the value of fault exposing potential FEP (S, t), if FEP ratio is zero it means that t_i does not cover any S_j.

FEP (S, t) = (mutant of S_j killed by each test t_i)/(Total number of mutant of S_j)

5.4.2 Average percentage of rate of faults of detection (APFD)

To measure the proposed algorithm performance, efficiency will be calculated by the fault detection rate. The performance of an algorithm is measured using the APFD metric shown in Eq. (3). APFD metric calculated as:

$$\text{APFD} = 1 - \frac{TC_1 + TC_2 + TC_3 + \cdots + TC_n}{n * m} + \frac{1}{2 * n} \quad (3)$$

where {TC_1,TC_2,TC_3,TC_4,… TC_n} represents set of test cases and m shows the total number of covered fault. TC_i shows first test case which detects the fault. Table 2 shows the test case which detects faults.

We used four approaches for ordering the test cases. Test suite S1 contains no-prioritized test cases. Test suite S2 contains a random test case. Test suite S3 contains reverse order. Test suite S4 is prioritized test cases using a genetic algorithm. We have considered five different types

Table 2 Test case covers the faults

Test case	Faults in program									
	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}
TC_1		×							×	
TC_2	×		×			×				×
TC_3										×
TC_4				×				×		
TC_5	×				×	×		×		
TC_6	×	×			×	×				
TC_7				×			×			
TC_8			×	×						
TC_9			×	×						
TC_{10}					×				×	

of object-oriented programs written in java to perform testing. Figure 7 shows the percentage of APFD for all four techniques.

ATM case study To show the performance of the proposed methodology, we have considered a case study. The case study we have considered in this paper is ATM taken from literature, which allows performing some basic banking operations like Deposit, Withdraw, Transfer, Balance, etc. Figure 8 shows use case diagram of ATM.

The model of ATM comprises of 17 classes, 23 operations, 28 attributes. The user's main function is to perform transactions from ATM. Types of transactions are performed such as Deposit money, Withdraw money, Transfer money and Inquiry. We have performed four different types of changes made to the original design. Change made between Version 1 and Version 2 is shown below in

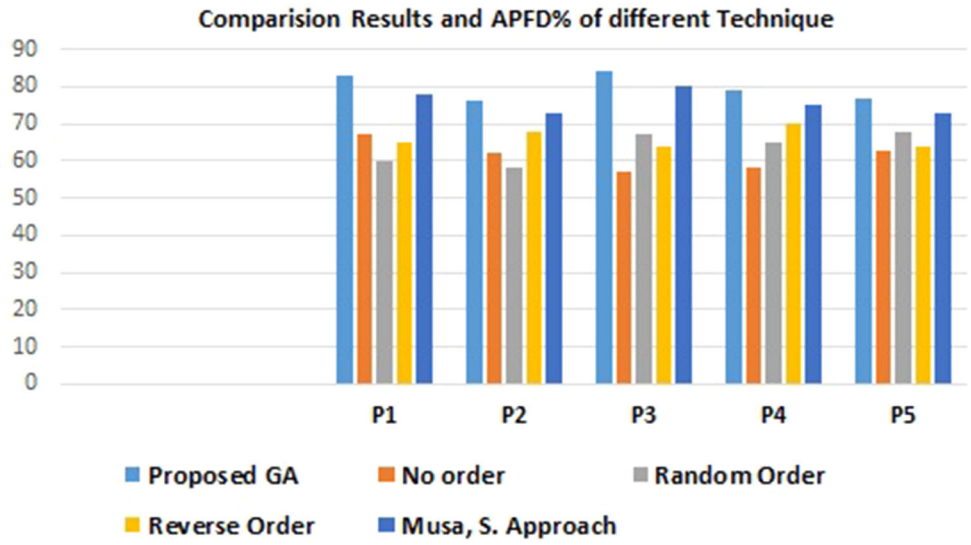


Fig. 7 Comparision result and APFD% of different technique

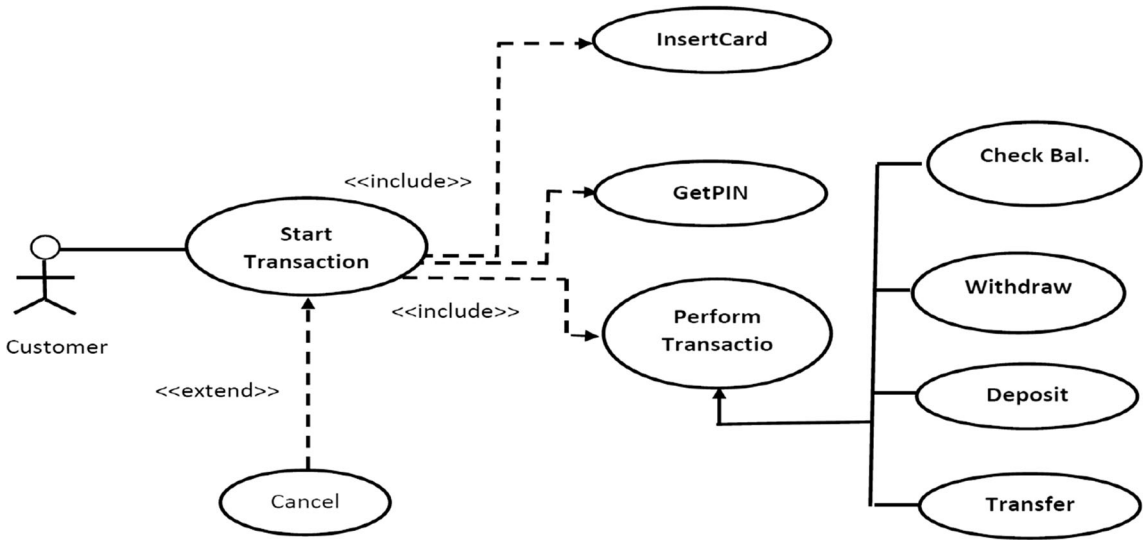


Fig. 8 Model for ATM

Table 3 Change made between Version 1 and Version 2

	Total (V1)	Added	Changed	Deleted	Total (V2)
Class	17	0	2	0	17
Method	23	3	3	0	26
Attribute	28	1	0	0	29

Table 4 Change made between Version 1 and Version 3

	Total (V1)	Added	Changed	Deleted	Total (V2)
Class	17	0	2	0	17
Method	23	0	3	0	23
Attribute	28	1	0	0	29

Table 5 Change made between Version 4

	Total (V1)	Added	Changed	Deleted	Total (V2)
Class	17	0	0	0	17
Method	23	4	0	2	25
Attribute	28	3	0	0	31

Table 3. Change made between Version 1 and Version 3 shown in Table 4. Change made between Version 4 is shown in Table 5.

6 Results analysis and comparisons

For empirical valuation, we used five application programs with different versions. These programs are written in Java programming language. We have conducted experiments with five different kinds of the program to check the efficiency of the proposed algorithm for prioritization of test cases. Faults are introduced in programs then test cases detect the faults in the program. The efficiency of an algorithm is calculated using the APFD metric. The algorithm takes test cases as input with the faults to prioritize the test cases. The fitness function calculates the fault detection rate and maximum statement covered by the test cases. The fitness function finds the best solutions for a problem from the randomly generated population. We have compared this algorithm with random prioritization, reverse prioritization, no-prioritization, and GA prioritization techniques and with other proposed technique. The proposed algorithm shows the highest value of APFD compared to different technique shown in Fig. 7. The genetic algorithm gives an optimal solution from the desired population. Hence, fitness function helps to select the best suitable population for a given problem. In crossover operation, two individuals are recombined. In mutation operation, individuals are swapped randomly. Third, if any duplicate individual is present then remove it. Finally, an optimized solution is checked. If the final obtained solution is not optimized, then, all genetic operators are used again to produce the population of the test case. Compared to other technique the proposed technique achieves better result than other technique.

A benchmark program written in java programming using Eclipse IDE is used for experiment purpose shown in Table 6. The original programs are modified and fault is seeded into the program.

We have proposed a design and code based approach for test case selection and prioritization methodology for OOP to compare with existing technique1 (Rothermel and Harrold 1997) and (Musa et al 2014). We have considered five benchmark small programs and implemented and compared with our approach. Figure 9 shows the percentage value of statement coverage achieved by test cases of proposed approach and other technique. Our approach selects the fewer number of test cases to exercise the modified code. Shown in Fig. 10. This selected test case covers a 100% modified code. In Fig. 7 we have shown prioritization using the GA approach which produces a maximum percentage value of the proposed method. Prioritization approach is compared with no order, reverse order, random method and GA based approach. Effectiveness of our proposed approach depends not only on code coverage, the percentage of the selected test case to

Table 6 Benchmark Program

S. no	Bench mark program	Original program				Modified program			
	Name of program	Class	Method	Faults	Test case	Add class	Add method	Faults	Test case
1	Banking program	17	23	8	48	19	26	11	42
2	Library management system	18	25	19	39	23	34	26	45
3	Student information system	15	12	16	32	17	16	19	38
4	Online test software	8	17	8	18	10	19	12	26
5	Internet banking	9	19	17	19	12	23	25	29

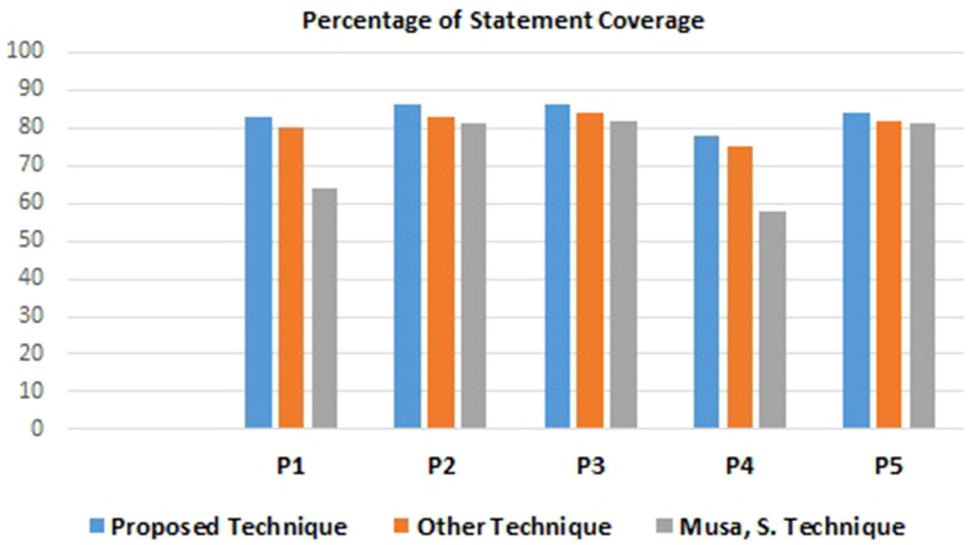


Fig. 9 Comparision of Percentage of statement coverage

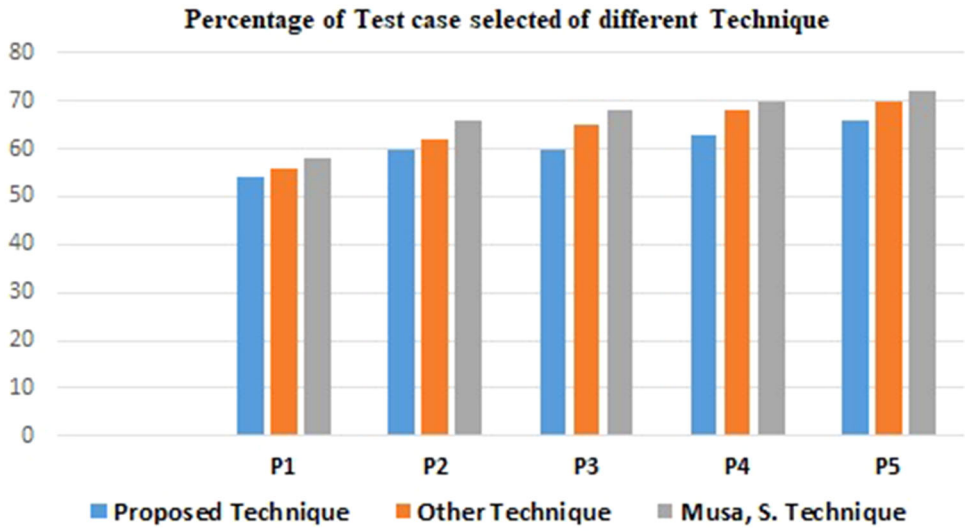


Fig. 10 Comparision of percentage of test case selected for different technique

cover modified code but also depends on how much fault is revealed by test cases. Related work is done by (Panigrahi and Mall 2012, 2013; Musa et al. 2015). We have proposed a test case selection method using UML and analysis of source code. First, logical changes are performed then identify the set of elements changed in UML and source code. Then performed testing for revised code to make sure that added or modified functionalities of the software were implemented correctly and there is no impact on unmodified functionalities. Changes are recognized by matching the original and changed version of code. Then GA is applied to test cases to prioritize in descending order. The work of (Musa et al. 2014) is similar to our proposed work. They select the change analysis from source code only.

Table 7 Comparision results of prioritization APFD % of different technique

S. no	Project no	Proposed GA prioritization APFD%	No order prioritization APFD%	Random order APFD%	Reverse order APFD%	Musa, S. approach prioritization APFD%
1	P1	83	67	60	65	78
2	P2	76	62	58	68	73
3	P3	84	57	67	64	80
4	P4	79	58	65	70	75
5	P5	77	63	68	64	73

Table 8 Experimental Results of Percentage of Fault detection rate

S. no	Project no.	Proposed GA (FDR)	Author Musa, S. approach (FDR)	Without GA (FDR)
1	P1	76	65	54
2	P2	83	73	60
3	P3	87	62	65
4	P4	88	77	50
5	P5	84	74	45

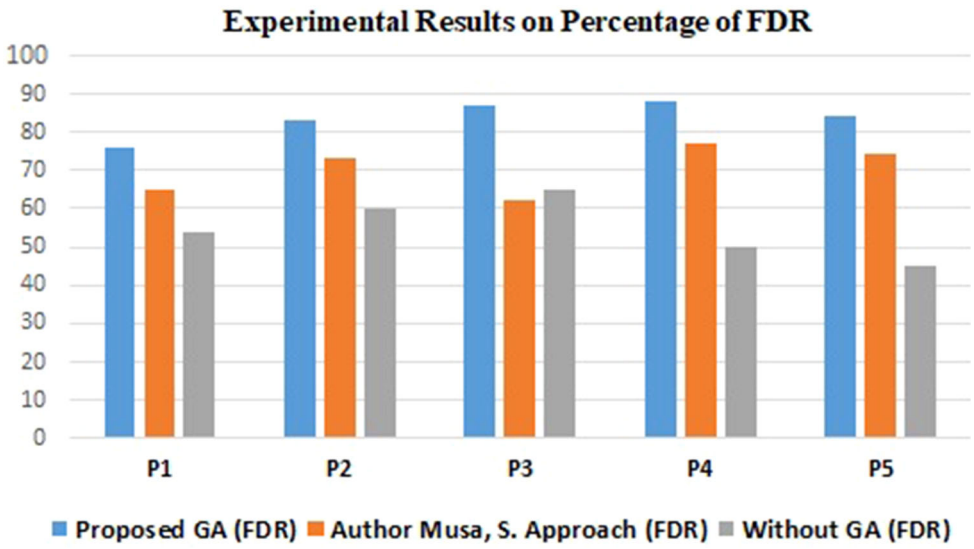


Fig. 11 Experiment results of percentage of FDR

Their method does not consider design level change. Our approach considers various object oriented features. We have considered design label and code based change impact which helps to detect fault at early stage. We have achieved maximum APFD value.

The proposed result and other proposed results are shown in Table 7. Tables 7 and 8 show the results of the proposed approach and other approaches for the benchmark programs. Table 7 shows proposed prioritization technique APFD value and compared result with other technique. Table 8 shows results on the fault detection rate percentage value in this case maximum no. of faults are detected by the test cases. Test case is prioritized based on maximum fault detection rate and maximum number of statement coverage by test case. Proposed GA approach detects maximum number of faults in the program compared with other approach. Experiment results of different technique of percentage of FDR is shown in Fig. 11. Proposed approach maximize the fault detection rate compared to other technique and selects minimum test case to covered modified code.

7 Conclusion

The proposed technique combines the analysis of the UML model and source code. We proposed the selection and prioritization of test cases using design and code based techniques. We used a graph model for intermediate representation for the object-oriented program to identify the change. The selection of the test case is done at design level using the UML model before implementation. The basic models are use case model and sequence diagram. Two models are constructed. Then models are compared to identify the change between these two models. Test cases are traversed on the modified model, and test cases that detect the affected model would be considered for regression testing. A genetic algorithm is applied for object-oriented software by identifying the test case that increases the fault detection rate in a program. This work reduces the cost of the regression test. Average Percentage of Faults Detected (APFD) metric is applied to validate the effectiveness of an algorithm. The proposed approach maximizes the value of APFD and maximum number of modified statement is covered. We have used a test case which covers maximum statement coverage and maximum fault coverage in the program. The APFD metric measures

the effectiveness and efficiency of an algorithm. The proposed algorithm produce less number of test cases to cover the modified code in regression testing compared with other technique.

Acknowledgements We are thankful to Department of Computer Science & Engineering, Birla Institute of Technology Mesra, Ranchi for providing the laboratory facilities to carryout this research work. We are heartly thankful to our Vice-Chancellor Prof. S. Konar and our Head of Department CSE Prof. V. Bhattacharya for their constant inspiration and encouragement. Without their support, this research article would not have been possible.

References

Agrawal AP, Arvinder K (2018) A comprehensive comparison of ant colony and hybrid particle swarm optimization algorithms through test case selection. In: Data engineering and intelligent computing. Springer, Singapore, pp 397–405

Al-Salami NMA (2009) Evolutionary algorithm definition. Am J Eng Appl Sci 2(4):789–795

Askarunisa MA, Shanmugapriya ML, Ramaraj DN (2010) Cost and coverage metrics for measuring the effectiveness of test case prioritization techniques. INFOCOMP 9(1):43–52

Badri L, Badri M, St-Yves D (2005) Supporting predictive change impact analysis: a control call graph based technique. In: 12th Asia-Pacific software engineering conference (APSEC'05). IEEE, Taipei, Taiwan

Batra G (2009) Model-based software regression testing for software components. In: Prasad SK (ed) Proceedings of the 3rd international conference on information systems, technology and management, vol 31. Springer, Berlin

Biswas S, Mall R, Satpathy M, Sukumaran S (2011) Regression test selection techniques: a survey. Informatica 35(3):289–321

Black S (2008) Deriving an approximation algorithm for automatic computation of ripple effect measures. Inf Softw Technol 50(7–8):723–736

Bohner SA (1996) Impact analysis in the software change process: the year 2000 perspective. Inicsm 96:42–51

Briand LC, Labiche Y, O'Sullivan L, Sówka MM (2006) Automated impact analysis of UML models. J Syst Softw 79(3):339–352

Byson NA (1995) A goal programming method for generating priority vectors. J Oper Res Soc 46(5):641–648

Chu AT, Kalaba RE, Spingarn K (1979) A comparison of two methods for determining the weights of belonging to fuzzy sets. J Optim Theory Appl 27(4):531–538

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 Test Verif Reliab 25(4):371–396

Engstrom E, Skoglund M, Runeson P (2008) Empirical evaluations of regression test selection techniques: a systematic review. In: Proceedings of the second ACM-IEEE international symposium on empirical software engineering and measurement, pp 22–31

Fahad M, Nadeem A (2008) A survey of UML based regression testing. In: Shi Z, Mercier-Laurent E, Leake D (eds) Intelligent information processing IV IIP 2008. IFIP – the international federation for information processing, vol 288. Springer, Boston, pp 200–210

Fang C, Chen Z, Wu K, Zhao Z (2014) Similarity-based test case prioritization using ordered sequences of program entities. Softw Qual J 22(2):335–361

Graves TL, Harrold MJ, Kim J-M, Porter A, Rothermel G (2001) An empirical study of regression test selection techniques. ACM Trans Softw Eng Methodol (TOSEM) 10(2):184–208

Gupta R, Harrold MJ, Soffa ML (1996) Program slicing-based regression testing techniques. Softw Test Verif Reliab 6(2):83–111

Hao D, Zhang L, Zhang L, Rothermel G, Mei H (2014) A unified test case prioritization approach. ACM Trans Softw Eng Methodol (TOSEM) 24(2):10–31

Harrold M, Soffa M (1989) Interprocedual data flow testing. ACM SIGSOFT Softw Eng Notes 14(8):158–167

Harrold MJ, Jones JA, Li T, Liang D, Orso A, Pennings M, Sinha S, Spoon SA, Gujarathi A (2001) A regression test selection for Java software. In: Proceedings of the 16th ACM SIGPLAN conference on object oriented programming systems languages and applications. ACM, New York, pp 312–326

Hewitt J, Rilling JA (2006) Light-weight proactive software change impact analysis using use case maps. In: Proceedings of the international workshop on software evolvability. IEEE, Budapest, Hungary, pp 41–46

Holland JH (1973) Genetic algorithms and the optimal allocation of trials. SIAM J Comput 2(2):88–105

Holland JH (1975) Adaptation in natural and artificial systems Ann Arbor. Univ Michigan Press 1:975

Horwitz S, Reps T, Binkley D (1990) Interprocedural slicing using dependence graphs. ACM Trans Program Lang Syst (TOPLAS) 12(1):26–60

Huang YC, Huang CY, Chang JR, Chen TY (2010) Design and analysis of cost-cognizant test case prioritization using genetic algorithm with test history. 2010 IEEE 34th annual computer software and applications conference. IEEE, Seoul, South Korea, pp 413–418

Iqbal MZZ, Malik ZI, Nadeem A (2007) An approach for selective state machine based regression testing. In: Proceedings of the 3rd international workshop on advances in model-based testing. ACM, New York, pp 44–52

Kim S, Clark JA, McDermid JA (2000) Class mutation: mutation testing for object-oriented programs. In: Proc Net ObjectDays. Net Objects, Erfurt, Germany, pp 9–12

Kundu D, Samanta D (2009) A novel approach to generate test cases from UML activity diagrams. J Object Technol 8(3):65

Larsen L, Harrold MJ (1996) Slicing object-oriented software. In: Proceedings of IEEE 18th international conference on software engineering, pp 495–505

Law J, Rothermel G (2003) Incremental dynamic impact analysis for evolving software systems. In: Proceedings of the international symposium on software reliability engineering. IEEE, Denver, CO, USA, pp 430–441

Le Traon Y, Jeron T, Jezequel JM, Morel P (2000) Efficient OO integration and regression testing. IEEE Trans Reliab 49(1):12–25

Li Z, Harman M, Hierons RM (2007) Search algorithms for regression test case prioritization. IEEE Trans Softw Eng 33(4):225–237

Lu Y, Lou Y, Cheng S, Zhang L, Hao D, Zhou Y, Zhang L (2016) How does regression test prioritization perform in real-world software evolution. In: IEEE/ACM 38th international conference on software engineering (ICSE). IEEE, Austin, TX, USA, pp 535–546

Luo Q, Moran K, Poshyvanyk D (2016) A large-scale empirical comparison of static and dynamic test case prioritization techniques. In: 24th ACM SIGSOFT international symposium on foundations of software engineering. ACM, New York, pp 559–570

Maia MCO, Bittencourt RA, DeFigueiredo JCA, Guerrero DDS (2010) The hybrid technique for object-oriented software change

impact analysis. In: 14th European conference on software maintenance and reengineering (CSMR). IEEE, Madrid, Spain, pp 252–255
Malishevsky AG, Ruthruff JR, Rothermel G, Elbaum S (2006) Cost-cognizant test case prioritization. Technical Report TR-UNL-CSE-2006-0004, University of Nebraska-Lincoln
Mansour N, Salem H (2006) Ripple effect in object-oriented programs. J Comput Methods Sci Eng 6(5):23–32
Marchetto A, Islam MM, Asghar W, Susi A, Scanniello G (2016) A multi-objective technique to prioritize test cases. IEEE Trans Softw Eng 42(10):918–940
Musa S, Sultan ABM, Ghani AABA, Baharom S (2014) A regression test case selection and prioritization for object-oriented programs using dependency graph and genetic algorithm. Res Inven Int J Eng Sci 4(7):54–64
Musa S, Sultan ABM, Abd-Ghani AAB, Salmi B (2015) Regression test cases selection for objectoriented programs based on affected statements. Int J Softw Eng Appl 9(10):91–108
Najumudheen ESF, Mall R, Samanta D (2009) A dependence graph-based test coverage analysis technique for object-oriented programs. In: 2009 sixth international conference on information technology: new generations, pp 763–768
Panda S, Munjal D, Mohapatra DP (2016) A slice-based change impact analysis for regression test case prioritization of object-oriented programs. Adv Sofw Eng. https://doi.org/10.1155/2016/7132404
Panigrahi CR, Mall R (2012) A hybrid regression test selection technique for object-oriented programs. Int J Softw Eng Appl 6(4):17–34
Panigrahi CR, Mall R (2013) An approach to prioritize the regression test cases of object-oriented programs. CSI Trans ICT 1(2):159–173
Pilskalns O, Uyan G, Andrews A (2006) Regression testing uml designs. In: 2006 22nd IEEE international conference on software maintenance, pp 254–264
Rothermel G, Harrold MJ (1994a) A framework for evaluating regression Test Selection Techniques. In: Proceeding of the 16th international conference on software engineering, ICSE 1994. IEEE, Sorrento, Italy, pp 201–210
Rothermel G, Harrold MJ (1994b) Selecting regression tests for object-oriented software. ICSM 94:14–25
Rothermel G, Harrold MJ (1996) Analyzing regression test selection techniques. IEEE Trans Softw Eng 22(8):529–551
Rothermel G, Harrold MJ (1997) A safe, efficient regression test selection technique. ACM Trans Softw Eng Methodol (TOSEM) 6(2):173–210
Rothermel G, Harrold MJ, Dedhia J (2000) Regression test selection for C++ software. J Softw Test Verif Reliab 10(2):77–109
Rovegard P, Angelis L, Wohlin C (2008) An empirical study on views of the importance of change impact analysis issues. IEEE Trans Softw Eng 34:516–530
Sabharwal S, Sibal R, Sharma C (2011) A genetic algorithm based approach for prioritization of test case scenarios in static testing. 2nd international conference on computer and communication technology (ICCCT). IEEE, Allahabad, India, pp 304–309
Shiri M, Hassine J, Rilling JA (2007) A requirement level modification analysis support framework. In: Third international IEEE workshop on software evolvability 2007. IEEE, Paris, France, pp 67–74
Singh Y, Kaur A, Suri B (2010) Test case prioritization using ant colony optimization. ACM SIGSOFT Softw Eng Notes 35(1):940
Skoglund M, Runeson P (2005) A case study of the class firewall regression test selection technique on a large scale distributed software system. In: 2005 international symposium on empirical software engineering. IEEE, p 10
White L, Jaber K, Robinson B (2005) Utilization of extended firewall for object-oriented regression testing. In: Proceedings of the 21st IEEE international conference on software maintenance. IEEE, Budapest, Hungary, pp 695–698
Yadav DK, Dutta S (2016) Test case prioritization technique based on early fault detection using fuzzy logic. 3rd IEEE international conference on computing for sustainable global development (INDIACom). IEEE, New Delhi, India, pp 1033–1036
Yadav DK, Dutta S (2017) Regression test case prioritization technique using genetic algorithm. In: Sahana S, Saha S (eds) Advances in computational intelligence Advances in intelligent systems and computing, vol 509. Springer, Singapore, pp 33–140
Yadav DK, Dutta S (2019a) Test case prioritization using clustering approach for object oriented software. Int J Inf Syst Model Des (IJISMD) 10(3):92–109
Yadav DK, Dutta SK (2019b) Test case prioritization based on early fault detection technique. Recent Patents Comput Sci. https://doi.org/10.2174/2213275912666190404152603 **(ISSN: 1874-4796 (Online))**