IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661, p- ISSN: 2278-8727Volume 16, Issue 3, Ver. IV (May-Jun. 2014), PP 38-47
www.iosrjournals.org

Regression Test Case Selection &PrioritizationUsing Dependence Graph and Genetic Algorithm

Samaila Musa[1], Abu Bakar M. d. Sultan[2], Abdul Azim Abd Ghani[3], Salmi Baharom[4]

[1] *(Department of Information System, Faculty of Computer Science & Information Technology, University Putra Malaysia)*
[2]*(Department of Information System, Faculty of Computer Science & Information Technology, University Putra Malaysia)*
[3] *(Department of Information System, Faculty of Computer Science & Information Technology, University Putra Malaysia)*
[4] *(Department of Information System, Faculty of Computer Science & Information Technology, University Putra Malaysia)*

Abstract: *Regression testing is very important process in software maintenance. Unfortunately, it iscostly and time consumingto allow for the re-execution of all test cases during regression testing. The challenge in regression testing is the selection of best test cases from the existing test suite.This paper presents an evolutionary regression test case prioritization for object-oriented software based on extended system dependence graph model of the affected program using genetic algorithm. The approach is based on optimization of selected test case from dependency analysis of the source codes. The goal is to identify changes in a method's body due to data dependence, control dependence and dependent due to object relation such as inheritance and polymorphism, select the test cases based on affected statements and ordered them based on their fitness by using GA.The number of affected statements determined how fit a test case is good for regression testing. A case study is reported to provide evidence of the feasibility of the approach and its benefits in increasing the rate of fault detection and reduction in regression testing effort compared with retest-all. It was shown that our approach needs 30% of the test cases to cover all the faults, while 80% is needed to cover all the faults using retest-all, which is time consuming and costly.*

Keywords: *Evolutionary algorithm, genetic algorithm, regression testing, regression test case, regression test case prioritization, system dependence graph.*

I. Introduction

Software maintenance activity is an expensive phase account for nearly 60% of the total cost of the software production [1].Regression testing is an important phase in software maintenance activity to ensure that modifications due to debugging or improvement do not affect the existing functionalities and the initial requirement of the design [2] and it almost takes 80% of the overall testing budget and up to 50% of the cost of software maintenance [3].

Regression test selection is a way that test cases are selected from an existing test suite, that need to be rerun to ensure that modified parts behave as intended and the modification have not introduce sudden faults. Prioritization of test cases to be used in testing modified program means reduction in the cost associated with regression testing.Identifying test cases that exercised modified parts of the software is the main objective of regression test selection.The challenge in regression testing is the prioritization of selected test cases by identifying and selecting of best test cases from the selected test cases, and selecting good test cases will reduce execution timeand maximize the coverage of fault detection.

Problem definition:

Let P be a certified program tested with test suite T, and P' be a modified program of P. During regression testing of P', T and information about the testing of P with T are available for use in testing P'.

To solve the above problem, Rothermel and Harrold [3] have outlined a typical selective retest technique that: Identify changes made to P by creating a mapping of the changes between P and P', select a set T' subset of T that may reveal changes-related faults in P', use T' to test P', identify if any parts of the system have not been tested adequately and generate a new set of test case T'', use T'' to test P`, to establish the correctness of P` with respect to T``

Regression testing approach can be based on source code, i.e., code-based and based on design, i.e., design-based, many of the approaches were proposed by the researchers. The more safe and easy to make are the approaches that generate the model directly from the source code of the software.

Code-based object oriented regression test selection techniques select test cases for regression testing based on detail analysis of the source code. It uses recovered relationships between code parts and test cases that traverse them to locate test cases for retest when code is modified. The techniques are safe and precise when compared with design-based techniques; the issue of invisible changes (i.e., changes inside the method body) may not be detected at design level.

After the regression test case selection, the challenge in regression testing is the prioritization of selected test cases by identifying and selecting of best ones from them, and selecting these test cases will reduce execution time and maximize the coverage of fault detection. Regression test case prioritization tends to order test cases for execution during regression testing efficiently. Ordering of test cases that reveals faults earlier is the most important in test case execution. Test case prioritization techniques relied on the available surrogates for prioritization criteria, because the position and nature of the faults are not generally known in advance [4]. Regression test case prioritization is a technique in which test cases are ordered with the aim of detecting faults early in the test execution cycle. Because with multiple modifications and versions of the system, tests become larger in size and it cannot be executed all within limited time [5].

According to [6], most of the test cases prioritization techniques proposed are at the test suite level, with the aim of detecting faults as early as possible in the regression testing process, using a test suite used in the previous versions of the system.

In order to solve the above problem, researchers have proposed many code-based approaches[7,8,9,10] by identifying modifications in the level of source code, but the authors focus on the procedural-based programming which are not suitable in object-oriented programming widely today used in software development. Other researchers[11,12,13,14,15,16] address the issues of object-oriented programming but do not consider some basic concept of object-oriented features (such as inheritance, polymorphism, etc.,) as bases in identifying changes.

In [7], the test case selection approach is based on analysis of data and control dependencies among modules in the procedural program. The set of all modified modules in a program along with those modules which interact with the modified modules are considered as a firewall.

A class and a statement are considered as units of testing C++ program are presented in [8]. In their technique, firewall-based regression test selection technique for C++ program was proposed based on the classes. The dependencies between various elements of a C++ program were represented in three models: Block Branch Diagram (BBD); represent the interface and the control structure of a method of a class, and the relationship of a class with other classes in the program. Object Relation Diagram (ORD); is a diagram that represent inheritance, aggregation and association relations, and captures the static dependencies between classes. Object State Diagram (OSD); is designed to capture the dynamic behaviour of class.

In [9], instead of considering a class and a statement as unit of testing [4], the authors considered a method as the unit of testing C++ program. This technique deals with modification at method level and with the aim of identify affected methods. The techniques defined all set of affected classes that need to be retested as firewall, and they select there all test cases which execute at least one class from within the firewall. These techniques are more efficient computationally and are favored for regression test selection of large program, but are not safe because the techniques do not select test case which may execute the effected class from outside the firewall.

A technique was proposed by [10] for regression test selection of object-oriented program by modeling the original program and modified program using Intermediate Program Dependency Graph (IPDG) for the application and Class Dependency Graph (ClDG) for both the original program and modified program for regression testing of modified and derived classes. The information of test coverage was used to associate the predicate and statement nodes of the ClDG models with each test case. Test cases are selected based on identified dangerous edges. Test cases which execute the set of these edges are assumed to be modification-revealing.

The algorithms to construct control flow graphs for a procedure/program and its modified version and use these graphs to select tests that execute changed code from the original test suite was presented [2]. The objective of the authors is to identify all nonobsolete tests in T(test suite) that execute changed code with respect to original and modified programs, i.e., to identify tests in T that (1) execute code that is new or modified for modified program or (2) executed code in original program that is no longer present in modified program.

In [11], an approach was proposed for regression test selection of C++ programs based on the analysis of the control flow representations of the original and modified programs. In this approach, the control flow of multi-function programs and object-oriented programs were represented using Inter-procedural Control Flow Graph (ICFG) and Class Control Flow Graph (CCFG). This approach is more efficient as compared to [2]. The novelty about this approach is that, the limitation of representing the control flow of information by CFG of only single method is overcome. The algorithm proposed in this approach traverse the graph model of original and modified programs and select relevant regression test case.

A safe regression test selection algorithm for Java software was presented in [12] that reduce the time required to carry out test as a result of selecting only the portion of test suite that may reveal a fault in the modified software. The technique is based on proposed approach by [11], and handles a lot of object-oriented features. They constructed Java Interclass Graph (JIG) for both original and modified programs, traversed them to identify dangerous edges, and from the coverage information obtained, test cases are selected that exercise the dangerous edges identified.

A general regression test case selection was presented in [13] based control flow analysis. This technique used Enhanced CFGs (ECFGs) which can be applied on most of the programming languages. Test Selection performs its comparison first at the class, then at the method level and finally at the node level. The algorithm produces sets of class pairs (C) by compares each class in P with the like named class in P'. For each pair of classes, the result of matching methods in the class in P with methods having the same signature in the class in P' is a set as method pairs (M). Then the algorithm constructs Enhanced CFGs (ECFGs) for the two methods and match nodes in the two ECFGs. Finally algorithm calls procedure compare, passing E and E' as parameters and return T'; the set of test cases selected. But the technique was only able to detect static change on the software.

A novel approach was presented by [14] named two-phase partitioning for regression test selection Java programs that is safe, precise, and yet scales to large systems. The proposed technique can efficiently reduce the regression testing effort and, thus, achieve considerable savings. The technique is to combine the effectiveness of regression test selection techniques that are precise but may be inefficient on large systems with the efficiency of techniques that work at a higher-level of abstraction and may, thus, be imprecise. The two-phase approach is (1) partitioning part that perform an initial high-level analysis, which identifies parts of the system to be further analyzed, and (2) in the selection part, an in-depth analysis of the identified parts, which selects the test cases in T(test suite) to rerun is performed. The selection part takes the partition identified by the first phase as input and performs edge-level test selection on the classes and interfaces in the partition. Control flow was used to analyse model.

An automated behavior regression testing technique was presented in [15], which generates a set of test inputs that are specifically targeted at the changed code, both the old and new versions of the code are executed with the generated test inputs, this resulted in a set of behavioral differences between the two versions, which can provide developers with details on how their changes have affected the behavior of the code.

An approach was presented in [16] based on the concept of Control Call Graphs (CCG), reduced form of Control Flow Graph (CFG). The technique is more precise and captures the structure of calls and related control than the traditional Call Graph (CG). However, it is difficult to extracting information about changes in the source code that may not have direct impact on the method call.

Researchers have also proposed various approaches[17,18,19,20,21,22,23,24,25,26,27]to address the issues related to prioritization.

A heuristic-based test case prioritization approach for object-oriented programs was presented in [18]. The technique was based on analysis of dependence model, as improvement for technique presented in [17]. The authors constructed a dependence model of a program from its source code, and when the program is modified, the model is updated to reflect the changes. The test cases that covered the affected nodes are selected for regression testing. The selected test cases are then prioritized by assigning initial weights of 1(one) to the affected nodes. But, the weight may have effect in the selection if the numbers of covered nodes are many in a test case, which may result in selection of test case that is not much relevant, which will result in increase of regression testing time.In [19], the authors described the prioritization of test cases in large software development environments. [20] proposed a prioritization technique based on historical execution data. A hybrid evolutionary algorithm approach was presented in [21] that automatically generate relevant unit test cases for object-oriented software by apply fitness function based on distance that accounts for runtime exceptions. [22] Performed an empirical study using several greedy algorithms.

An approach that assumes all test case costs and fault severities are the same was presented in [23]. They proposed ordering to produce a more satisfactory order, the cost-cognizant metric that incorporates varying test case costs and fault severities. They propose a cost-cognizant test case prioritization technique based on the use of historical records and a genetic algorithm. When all test costs and faults severities are uniform, the experimental results indicate that the proposed technique frequently yields a higher Average Percentage of Faults Detected per Cost, and also show that the proposed technique is also useful in terms of Average Percentage of Faults Detected per Cost when all test case costs and fault severities are uniform.

In [24], the authors present a new metric for assessing rate of fault dependency detection and an algorithm to prioritize test cases. The new metric was used to show the effectiveness of the prioritization compared with non-prioritized test case. The results prove that prioritized test cases are more effective in detecting dependency among faults.A new prioritizing approach using Genetic Algorithm to prioritize the regression test suite is introduced [25] that will order test cases dynamically on the basis of complete code coverage, also generate new test cases using PMX and cyclic crossover and analysis is done on the basis of process cost and test cost. The proposed approach is aim at reducing the number of test cases that need to be run after changes have been made to the program.

A hybrid Test Suite-Test Case Refinement Technique was presented in [26] to reduce regression test case pool size, reduce regression testing time, cost & effort and also ensure the quality of the engineered product The approach is a regression test case optimization technique that is a hybrid of Test Case Minimization based on specifications and Test Case Prioritization based on risk exposure.An approach of hierarchical for system test case prioritization based on requirements has been proposed in [27] that maps requirements on the system test cases. A value was analyses and assigns to each requirement based on a comprehensive set of twelve factors thereby prioritizing the requirements, mapped on the highly relevant module and then prioritized set of test cases. The effectiveness of their approach is analyse using a case study of income tax calculator software using existing as well as the proposed approach. The results indicate the efficacy of the proposed approach in terms of fault detection and severity early.

In this paper we present an evolutionary prioritized approach that will select best test cases from existing test suite T used to test the original program P by using Extended System Dependence Graph (ESDG) [28] as an intermediate to identify the changes in P, at statements level. Identification of changes using this kind of graph will leads to précised detection of changes. The changed statements will be used to identify affected statements, and test cases that execute the affected statements are selected for regression testing. The selected test cases are prioritized by using genetic algorithm in order to have a superior rate of fault detection. This approach will reduce the cost of regression testing by increasing the rate of faults detection and reducing the number of test cases to be used in testing the modified program.

The rest of this paper is organized as follows. In the next section, we provide the details of the methodology used. Section 3 describes our test selection frame work. Section 4presents the empirical results and analysis. Section 5 concludes this paper.

II. Methodology

In this section, we describe variousmethods and materials used in our approach.

1.1 Extended System Dependence Graph (ESDG)

Extended System Dependence Graph (ESDG) [28] is a graph that can represents control and data dependencies, and information pertaining to various types of dependencies arising from object-relations such as association, inheritance and polymorphism. Analysis at statement levels with ESDG model helps in identifying changes at basic simple statement levels, simple method call statements, and polymorphic method calls. The dependency graph is based on the approach presented in [28], used to model object-oriented programs and is an extension of System Dependence Graph (SDG) [29] used to model procedural programs.
ESDG is a directed, connected graph G = (V, E), that consist of set of V vertices and a set E of edges.

1.1.1 ESDG Vertices

A vertex v represents one of the four types of vertices, namely, statement vertices, entry vertices, parameter, and polymorphic vertices

1.1.1.1 Statement vertices

Are program statements present in the methods body. Statement vertices are of two types: call vertices and simple statement vertices. Call vertices are used to represent method call statements, and all other statements such as conditionals, loops and assignment in the program are represented by simple statement vertices.

1.1.1.2 Parameter vertices

These are used to represent parameters passing between a caller and callee methods. They are of four types: formal-in, formal-out, actual-in, and actual-out. Actual –in and actual-out vertices are created for each call vertex and create formal-in and formal-out vertices for each method entry vertex.

1.1.1.3 Entry vertices

Methods and classes have entry vertices. A class entry vertex and a method entry vertex represent an entry into a class and an entry into a method respectively.

1.1.1.4 Polymorphic choice vertex

It is used to represent dynamic choice among the possible bindings in a polymorphic call.

1.1.2 ESDG Edges

An edge e represent one of the six edges, namely, control dependence edges, data dependence edges, parameter dependence edges, method call edges, summary edges, and class member edges

1.1.2.1 Control dependence edge

It is used to represents control dependence relations between two statement vertices.

1.1.2.2 Data dependence edge

It is used to represents data dependence relations between statement vertices.

1.1.2.3 Call edge

It is used to connect a calling statement to a method entry vertex. It also connects various possible polymorphic method call vertices to a polymorphic choice vertex.

1.1.2.4 parameter dependence edge

It is used for passing values between actual and formal parameters in a method call. It is of two types: parameter-in and parameter-out edges.

1.1.2.5 Summary edge

It is used to represents the transitive dependence between actual-in actual-out vertices.

1.1.2.6 Class member edge

It is used to represents the membership relation between a class and its methods. It is used to connect a class entry vertex to a method entry vertex.

1.2 Genetic Algorithm

Many real life problems have been solved using evolutionary algorithms, and GA is one such evolutionary algorithm. Some of the real life problems where GA was applied are:For railway scheduling problem, the travelling salesman problem, the vehicle routing problem andField of software engineering problems.

Genetic Algorithm has emerged as optimization technique and search method. Problems being solved by GA are represented by a population of chromosomes as the solution to the problems. A chromosome can be string of binary digits, integer, real or characters, and each string that makes up a chromosome is called a gene. This initial population can be totally random or can be created manually using processes such as heuristic technique. The pseudo code of a basic algorithm for GA can be:

```
Initialize (population)
Evaluate (population)
While (stopping condition not satisfied){
        Selection (population)
        Crossover (population)
        Mutate (population)
        Evaluate (population)
        }
```

A GA has three operators that are applied on its population:

1.2.1 Selection

In selection the offspring producing individuals are chosen. The first step is fitness assignment.Each chromosome is evaluated in present generation to determine its fitness value using the following formula:

$$Ft(pi) = \sum_{j=1}^{k} n\left(t_j\right) * p_{ij} \qquad \text{equation 1}$$

Where Pij is the position of the test case j in the chromosome i, and
$n(t_j)$ is the number of affected nodes in the test case j.
Each individual in the selection pool receives a reproduction probability depending on the own objective value and the objective value of all other individuals in the selection pool. This fitness is used for the actual selection step afterwards, and it is the sum of all of the fitness values in a chromosome.

1.2.2 Crossover or Recombination

After selection, is to apply crossover operation to the selected chromosomes. It involves swapping of genes or sequence of bits in the string between two individuals.

1.2.3 Mutation

Mutation operation alters chromosomes in small ways to introduce new good traits.

1.3 Average percentage of rate of Faults Detection APFD

To evaluate the performance of regression test case ordering, researchers [14,15,16,17,27,30,31] used APFD metric. The APFD metric of program P and test suite T is given as:

$$\mathrm{APFD}(P,T) = 1 - \frac{(Tf_1+Tf_2+\cdots\ldots Tf_m)}{nm} + \frac{1}{2n} \qquad \text{equation 2}$$

Where,
m is the number of faults
n is the number of test cases
$(Tf_1 + Tf_2 + \cdots \ldots Tf_m)$ are the positions of the first test case in T that expose the faults 1,2, ….,m.

III. Regression Test Framework

This paper presents an approach for the selection of test cases T` from the existing test suite T to be used in testing the modified program P`, and prioritized the selected test cases in order that will increases their rate of detecting faults.Fig. 4illustrates the various activities of the test case selection framework.

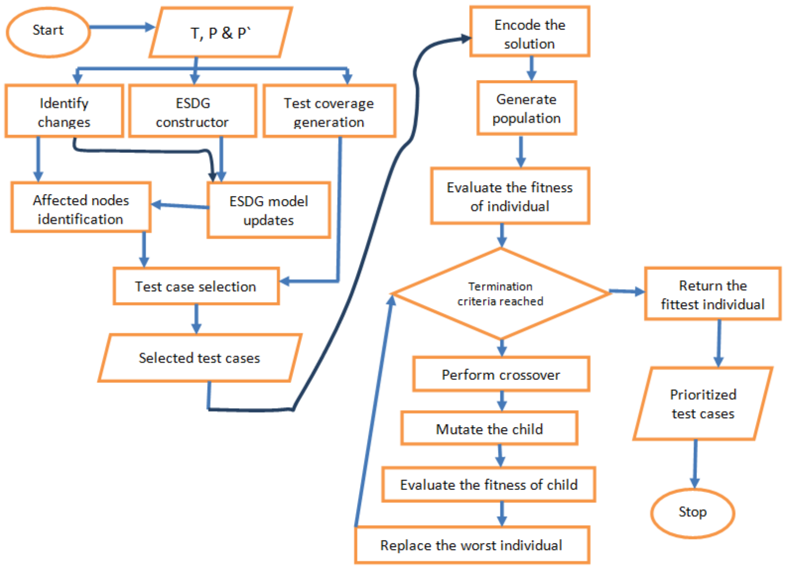

Figure 1 flowchart of our proposed approach.

1.4 Identify Changes

The changes between P and the modified program P` are identified in this step, via semantic analysis of the source code of the software. A file named "changed" will be used to store the identified statement level differences.

1.5 Test CoverageGeneration

Program P is instrumented at the statement levels. The code statements are executed with the original test suite T and to write traces for each test case in order to generate information pertaining to the specific statements that are executed for each test case.The generation of the test coverage information is perform once for a given program during one testing cycle, and the activity will not be repeated for the subsequent regression testing which will saved time. The information generated in this stage is saved in a file named "coverageInfo"for later use.

1.6 ESDG Model Constructor

ESDG model for the original program P is constructed using a technique similar to [15], and was described in section III.GraphViz will be used to represents our graph.

1.7 ESDG Model Updates

The model constructed for P is updated using information from changed file during each regression testing to make it correspond to the modified program P` and the updated ESDG model is denoted by M`.

1.8 Affected Statements identification

To identify the affected statements, a forward slice is constructed on the updated model M` using the information from changed file. Each changenode in changed file is used as slicing criterion todetermine the affected nodes in each statement. The affected nodes stored in changed file are used to identify the affected statements. The affected statements are statements that were affected directly by the modifications or as the result of control dependence or data dependence or dependent as a result of object relation such as inheritance and polymorphism on the affected node from the updated model M`, and denoted by”affectedStat”.

1.9 Test Case Selection

Test cases that execute the affected statements in the updated model M`are selected for regression testing, and donated as T`.

1.10Test Case Prioritization

To order the test cases selected in section 3.6, we follow the following steps:

1.10.1 Encode the solution

Create n population by encoding the test cases into integer numbers representations as P1 and reverse the order to have second parent/chromosomes as P2

1.10.2 Evaluate the fitness

The fitness of each parent/chromosome in the populationis evaluated using (1).

1.10.3 Selection

The chromosomes P1 and P2 are use as the initial population.

1.10.4 Crossover

Arandom integer number r is generated to serve as crossover point. The first rth genes of P1 will be copied to child C1, and copy the remaining genes from P2 that are not in C1.

1.10.5 Mutation

Mutate the child C1 using swap mutation technique. Generates two(2) random integer numbers, these numbers m1 and m2 serve as mutation points, simply swap their genes. Evaluate the fitness of the child C1 using (1).

1.10.6 Replace worst

This means replacing worst individual by the offspring in the current population.

1.10.7 Termination condition

The processes from step 3.7.4 to 3.7.6 will be repeated until maximum number of generation is reached. Then, the fittest individual is return as the best ordering.

IV. Empirical Results And Discussion

The empirical procedure of our proposed approach is: we generate the test suite T using path-based integration testing method for a given software/program, save the coverage information for each test case in T, construct the ESDG for the program, mutate the program P to P`, identify the changes between P and P`, update the ESDG model using the changes, identify the affected statements using the changed statements as slice criteria to perform forward slicing on the updated model, select test case T` from T that execute one of the affected statements, encode the solution into integer representation as first parent P1, reverse the order of P1 to be the second parent P2, evaluate the fitness of P1 and P2 using (1), perform crossover on the two parents to have a child C1, mutate the child, evaluate the fitness of C1 using (1), add the child to the population, remove the worst chromosome from the population, repeat crossover and mutation for a specific time and return the best chromosome.

We used a sample program of simple ATM with ten test cases {t1 t2 t3 t4 t5 t6 t7 t8 t9 t10}as shown in Table 1;T is the set of test cases in T, CoverageInform is the coverage information generated using procedure described in section 3.2, mutants are the changed statements at class and traditional levels, AffactNodes are the statements that are directly affected or dependents on the changed statements as described in section 3.5. We compare our results with retest-all technique, i.e., regression testing with T.

As shown in Table 1, there is increase in number of nodes in affected nodes column compared with the mutant’s column. This indicates that the forward slicing performed on the updated model produced more nodes that are dependent on the changed nodes/statements. Without using the affected nodes values, executing any of the test case 1, 2, or 8 will have 22.2% percentages of faults detection. The test cases will only detect five (5) out of nine (9) faults. But using affected nodes columns, executing test cases 1 first will cover33.3%, executing test case 2 first will cover 44.4%, and test case 8 will also cover 44.4%.

Table 1 Coverage Information of Test Suite T.

T	**CoverageInform**	**Mutants**	**AffactNodes**
T1	n2 n3 n4 n5 n6 n7 n8 n9 n10 n11 n12 n13 n14 n15 n16 n17 n18 n19 n20 n22 n23 n24 n25 n92 n93 n26 n103 n27 n88 n89 n94 n95 n28 n42 n43 n44 n45 n46 n47 n48 n49 n56 n62 n63	n27 n94	n27 n89 n94
T2	n2 n3 n4 n5 n6 n7 n8 n9 n10 n11 n12 n13 n14 n15 n16 n17 n18 n19 n20 n22 n23 n29 n30 n31 n32 n33 n97 n34 n66 n67 n80 n83 n86 n103 n97 n87 n35 n42 n43 n44 n45 n46 n47 n48 n49 n56 n62 n63	n83n97	n83 n86 n87 n97
T3	n2 n3 n4 n5 n6 n7 n8 n9 n10 n11 n12 n13 n14 n15 n16 n17 n18 n19 n20 n22 n23 n29 n36 n37 n64 n65 n106 n107 n38 n42 n43 n44 n45 n46 n47 n48 n49 n56 n62 n63	Nil	Nil
T4	n2 n3 n4 n5 n6 n7 n8 n9 n10 n11 n12 n13 n14 n15 n16 n17 n18 n19 n20 n22 n23 n29 n36 n37 n64 n65 n106 n107 n38 n42 n43 n44 n45 n46 n47 n48 n49 n50 n56 n62 n63	Nil	Nil
T5	n2 n3 n4 n5 n6 n7 n8 n9 n10 n11 n12 n13 n14 n15 n16 n17 n18 n19 n20 n22 n23 n29 n36 n39 n40 n41 n42 n43 n44 n51 n52 n53 n54 n55 n56 n62 n63	Nil	Nil
T6	n2 n3 n4 n5 n6 n7 n8 n9 n10 n11 n12 n13 n14 n15 n16 n17 n18 n19 n20 n21 n42 n43 n44 n45 n46 n47 n48 n56 n62 n63	Nil	Nil
T7	n2 n3 n4 n5 n6 n7 n8 n9 n10 n11 n57 n59 n60 n61 n62 n63	Nil	Nil
T8	n2 n3 n4 n5 n6 n7 n8 n9 n10 n11 n12 n13 n14 n15 n16 n17 n18 n19 n20 n22 n23 n29 n30 n31 n32 n33 n97 n34 n66 n67 n80 n83 n84 n85 n35 n42 n43 n44 n45 n46 n47 n48 n49 n56 n62 n63	n83 n97	n83 n84 n85 n97
T9	n2 n3 n4 n5 n6 n7 n8 n9 n10 n11 n12 n13 n14 n15 n16 n17 n18 n19 n20 n22 n23 n29 n30 n31 n32 n33 n97 n34 n66 n67 n80 n81 n82 n35 n42 n43 n44 n45 n46 n47 n48 n49 n56 n62 n63	n97	n97
T10	n2 n3 n4 n5 n6 n7 n8 n9 n10 n11 n12 n13 n14 n15 n16 n17 n18 n19 n20 n22 n23 n29 n30 n31 n32 n33 n97 n34 n66 n67 n103 n68 n69 n70 n35 n42 n43 n44 n45 n46 n47 n48 n49 n56 n62 n63	n97	n97

Table 2 presents the results of selected test cases that execute one of the affected statements as described in section 3.6. T` is the selected test cases, NOfAffStat is the number of affected statements for each selected test case T`.

Table 2 Selected Test Cases

T`	**NOfAffStat**
T1	3
T2	4
T8	4
T9	1
T10	1

We used GA to order the selected test cases T` as described in section 3.7. Our approach ordered the test cases as {t2 t8 t1 t9 t10 t3 t4 t5 t6 t7}, while retest-all ordering is {t1 t2 t3 t4 t5 t6 t7 t8 t9 t10}. The number of test cases is 10, and number of faults is 9 (f1 f2 f3 f4 f5 f6 f7 f8 f9 representing the faults n27 n83 n84 n85 n86 n87 n89 n94 n97 respectively). This is represented in Table 3.

For our proposed approach ordering of the test cases { t2 t8 t1 t9 t10 t3 t4 t5 t6 t7}, using (2) the APFD from Table 1 is:

$$\mathrm{APFD}(\mathrm{T},\mathrm{P}) = 1 - \frac{(3+1+2+2+1+1+3+3+1)}{10*9} + \frac{1}{2*10} = 0.8611$$

For retest-all ordering of the test cases {t1 t2 t3 t4 t5 t6 t7 t8 t9 t10} using (2) the APFD from Table I is:

$$\mathrm{APFD}(\mathrm{T},\mathrm{P}) = 1 - \frac{(1+2+8+8+2+2+1+1+2)}{10*9} + \frac{1}{2*10} = 0.75$$

Fig. 1 describes the above results to shows the differences between our approach and retest-all.

Table 3: The Faults Detected by the Test Cases

Testcases Faults	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
F1	x									
F2		x						x		
F3								x		
F4								x		
F5		x								
F6		x								
F7	x									
F8	x									
F9		x						x	x	x

The results of APFD for our approach and retest-all are presented in Fig. 2.The comparison is between our proposed approach and retest-all using T. From Table 2 and Fig.2, it is observed that our approach identify the faults at early stage. The APFD is better in our approach when compared to retest-all; this means that our proposed approach is more effective in term of rate of faults detection.

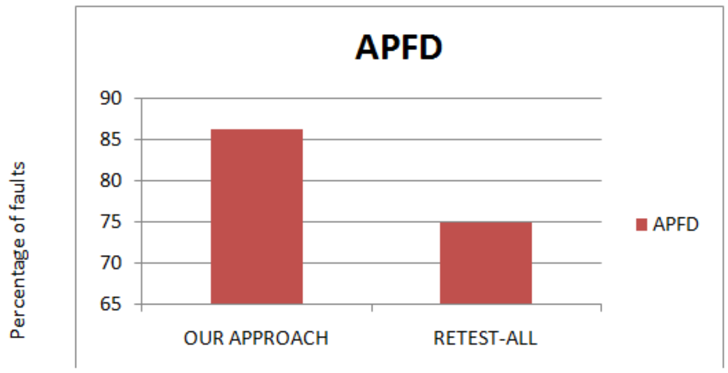

Figure 2. APFD of the approaches

In Fig. 3, we plot percentage of faultsdetected against percentage of test cases executed. From the result, we identified that our approach needs less test cases to reveal all faults compared to retest-all. It is observed that our approach needs 30% of the test cases to find out all the faults. But using retest-all there is need of 80% of the test cases to find out all the faults.

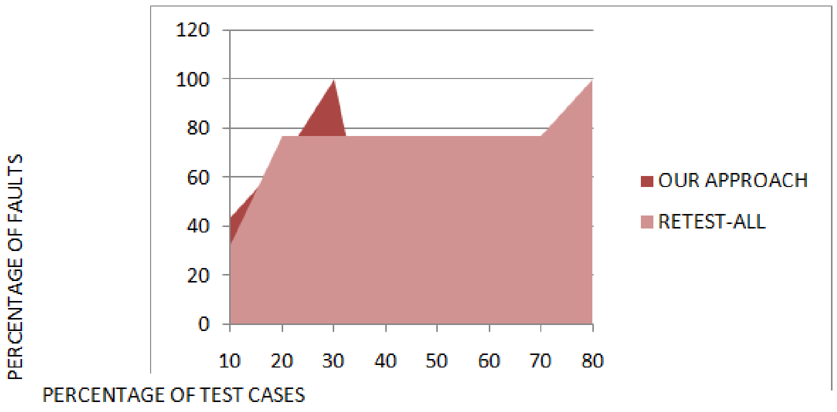

Figure 3percentages of test cases and faults.

We note that our proposed approach assumes that the ESDG are updated in a timely way, every time changes are introduced into the programs. This assumption becomes a limitation for our approach. Another limitation is that we assumed that all test costs and faults severities are uniform.

V. Conclusion

A GA prioritized test case framework has been proposed in our approach that ordered selected test cases T` from test suite T to be used for rerun in regression testing. The approach used extended system dependence graph (ESDG) to identify changes at statement level of source code, store the changes in a file named changed, and generate coverage information for each test case from the source code. The changedinformation are used to identify the affected statements, and test cases are identify that will be rerun in regression testing based on the affected statements.The selected test cases will be prioritized using genetic algorithm in order to increases their rate of faults detection.The technique cover the different important issues that regression testing strategies need to address: change identification, test selection, test execution and test suite maintenance.

The results indicate that our proposed ordering of test cases detect all the faults in early stage when compared to retest-all of the test cases. This shows that, our ordering of test case will reduce the regression testing time compared with retest-all of the test cases.

References

[1] S. P. Roger, Software engineering: A practitioner's approach (Fifth Edition. McGraw-Hill Publisher, New York, America, 2001).

[2] G. Rothermel and M. Harrold, Selecting regression tests for object-oriented software, International Conference on Software Maintenance, Victoria, CA, September 1994, 14–25.

[3] G. Rothermel, , M.J. Harrold, A safe, efficient regression test selection technique, ACM Transactions on Software Engineering Methodology, 6(2), 1997, 173–210.

[4] A. Orso, N. Shi, and M. Harrold. Scaling regression testing to large software systems,ACM SIGSOFT, Twelfth International Symposium on Foundations of Software Engineering, 29(6), 2004, 241–251.

[5] J. M. Kim, and A. Porter, A history-based test prioritization technique for regression testing in resource constrained environments, In Proceedings of the 24th International Conference on Software Engineering, 2002, New york, NY, USA, 119-129.

[6] L. S. de Souza, R. B. C. Prudêncio, F. de A. Barros, E. H. da S. Aranha, Search based constrained test case selection using execution effort, Expert Systems with Applications 40(12), 2013, 4887–4896.

[7] H. Leung and L. White, A firewall concept for both control-flow and data-flow in regression integration testing, In Proceedings of the Conference on Software Maintenance, Orlando, FL, 1992, 262–270.

[8] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen, On regression testing of object oriented programs, Journal of Systems and Software, 32(1), 1996, 21–40.

[9] Y. Jang, M. Munro, and Y. Kwon, An improved method of selecting regression tests for C++ programs, Journal of Software Maintenance: Research and Practice, 13(5), 2001, 331–350.

[10] W. Lee, J. Khaled, and R. Brian, Utilization of extended firewall for object-oriented regression testing, Proceedings of the 21st IEEE International Conference on Software Maintenance(ICSM'05), Cleveland, OH, USA, 2005, 1-4.

[11] G. Rothermel, M. Harrold, and J. Dedhia. Regression test selection for C++ software, Software Testing, Verification and Reliability, 10(2), 2000, 77–109.

[12] M. J. Harrold, J. A. Jones, T. Li, and D. Liang, Regression test selection for java software, ACM SIGPLAN Notices, 36(11), 2001, 312-326.

[13] W. S. A. El-hamid, S. S. El-etriby, and M. M. Hadhoud, Regression Test selection technique for multi-programming language, Faculty of Computer and Information, Menofia University, Shebin-Elkom, 32511, Egypt (2009).

[14] A. Beszedes, T. Gergely,, L. Schrettner, J. Jasz,, L. Lango, T. Gyimothy, Code coverage-based regression test selection and prioritization in webkit, 28th IEEE International Conference on Software Maintenance (ICSM 2012), 46-55.

[15] W. Jin, A. Orso, and T. Xie, Automated Behavioral regression testing, 2010 Third International Conference on Software Testing, Verification and Validation, Washington DC, USA, 2010, 137-146.

[16] N. Frechette, L. Badri, and M. Badri, Regression Test reduction for object-oriented software: A control call graph based technique and associated tool, 2013 International Scholarly Research Network Software engineering (ISRN Software Engineering2013), 1-10.

[17] C. Panigrahi, and R. Mall, An approach to prioritize regression test cases of object-oriented programs, JCSI Trans ICT (Springer) 2013, doi:10.1007/s40012-013-0011-7, 1-15.

[18] C. Panigrahi, and R. Mall, A heuristic-based regression test case prioritization approach for object-oriented programs, Innovations in Systems and Software Engineering (Springer) 2013, doi: 10.1007/s11334-013-0221-z, 1-9

[19] A. Srivastava and J. Thiagarajan, Effectively prioritizing tests in development environment, Proceedings of the International Symposium of Software Testing and Analysis, Rome, 2002, 97-106.

[20] S. Wappler and J. Wegener,Evolutionary Unit Testing Of Object-Oriented Software Using A Hybrid Evolutionary Algorithm, 2006 IEEE Congress on Evolutionary Computation, Sheraton Vancouver Wall Centre Hotel, Vancouver, BC, Canada, 2006, 851-858.

[21] Z. Li, M. Harman, and R. M. Hierons, Search algorithms for regression test case prioritization, IEEE Transaction on Software Engineering, 33(4), 2007, 225-237.

[22] Y.C. Huang, C.Y. Huang, J.R. Chang and T.Y. Chen, Design and Analysis of Cost-Cognizant Test Case Prioritization Using Genetic Algorithm with Test History, 2010 IEEE 34th Annual Computer Software and Applications Conference, Seoul, Korea. 2010, 413-418.

[23] M. I. Kayes, Test Case Prioritization for Regression Testing Based on Fault Dependency, 3rd International Conference on Electronics Computer Technology (ICECT), 2011 IEEE, kanyakumari, India, 2011, 48-52.

[24] Suman, and Seema, A Genetic Algorithm for Regression Test Sequence Optimization, International Journal of Advanced Research in Computer and Communication Engineering, 1(7), 2012, 478-481.

[25] A.S.A. Ansari, K.K. Devadkar, and P. Gharpure,Optimization of test suite- test case in regression test,2013 IEEE International Conference on Computational Intelligence and Computing Research, Enathi, India, 1-4.

[26] H. Kumar, V. Pal, N. Chauhan, A hierarchical system test case prioritization technique based on requirements, 13th Annual International Software Testing Conference, Bangalore, India, 2013, 4–5,

[27] S. Raju, and G. V. Uma, Factors Oriented test case prioritization technique in regression testing using genetic algorithm, European Journal of Scientific Research, 74(3), 2012, 389-402.

[28] L. Larsen, and M. Harrold, Slicing object-oriented software, In Proceedings of 18th IEEE International Conference on Software Engineering, Berlin, 1996, 495-505.

[29] S. Horwitz, T. Reps, and D. Binkley, Interprocedural slicing using dependence graphs, ACM Transactions on Programming Languages and Systems, 12(1), 1990, 26–60.

[30] H. Raperia and S. Srivastava, An Emperical Approach for Test Case Prioritization, International Journal of Scientific & Engineering Research, 4(3), 2013, 1-5.

[31] R. Gupta and A. K. Yadav, Study of test case prioritization technique using APFD, International Journal of Computers & Technology, 10(3), 2013, 1475-1481.