Abstract—Regression testing takes almost half of the cost of software maintenance, but it is very important process in software testing. Unfortunately, it may be costly to 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 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 will be 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.

Keywords—Evolutionary algorithm, regression testing, regression test case prioritization, and 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 time and maximize the coverage of fault detection.

Regression testing approach can be based on source code, i.e., code-based and based on design, i.e., design-based, many of them 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.

Researchers have proposed many procedural code-based approaches [3, 4, 5] by identifying modifications in the level of source code. Other researchers [2, 6, 7, 8, 9, 10, 11] address the issues of object-oriented programming but do not consider some basic concept of object-oriented features (such as inheritance, polymorphism, etc.), as a bases in identifying changes.

Researchers have proposed various approaches [13, 14, 15, 16] to address the issues related to prioritization.

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 Dependence Graph (ESDG) [12] as an intermediate to identify the changes in P, at statements level. 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 will be prioritized by using genetic algorithm in order to have a superior rate of fault detection when compared to rates of randomly and reversed prioritized test cases. 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 regression testing. Section 3 describes Extended System Dependence Graph (ESDG). In section 4, we introduce GA. In section 5, we present our test selection prioritization technique. Section 6 concludes this paper.
II. REGRESSION TESTING

Regression testing is a software testing activity normally conducted after software is changed, and its helps not only to ensure that changes due to debugging or improvement do not affect the existing functionalities but also the changes do not affect the initial requirement of the design. Regression test selection is an activity that select test cases 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.

Regression test selection technique will help in selecting a subset of test cases from the test suite. The easiest way is that, the tester simply executes all of the existing test cases to ensure that the new changes are harmless and is referred as retest-all method [9]. It is the safest technique, but it is possible only if the test suite is small in size. The test case can be selected at random to reduce the size of the test suite. But most of the test cases selected randomly can result in checking small parts of the modified software, or may not even have any relation with the modified program. Regression test selection techniques will be an alternative approach.

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'.
- Use the result of the above step to select a set T' subset of T that may reveal changes-related faults in P'.
- Use T' to test P', to establish the correctness of P' with respect to T'.
- 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''.

III. DEPENDENCY GRAPH

In this section, we describe the dependency graph based on the approach presented in [12]. Extended System Dependence Graph (ESDG) 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.

ESDG is a directed, connected graph G = (V, E), that consist of set of V vertices and a set E of edges.

A. ESDG Vertices

A vertex v represents one of the four types of vertices:

- Statement vertices: Are used to represent program statements present in the methods body.
- Parameter vertices: Are used to represent parameter passing between a caller and callee method. 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.
- Entry vertices: Methods and classes have entry vertices.
- Polymorphic choice vertex: it is used to represent dynamic choice among the possible bindings in a polymorphic call.

B. ESDG edges: An edge e represent one of the six edges:

- Control dependence edge: It is used to represents control dependence relations between two statement vertices.
- Data dependence edge: It is used to represents data dependence relations between statement vertices.
- Call edge: 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.
- 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.
- Summary edge: It is used to represents the transitive dependence between actual-in actual-out vertices.
- Class member edge: It is used to represents the membership relation between a class and its methods.

Figure 1 represent the different graphical symbols used represent the different types of vertices and edges.

![Graphical symbols used to represent the different types of vertices and edges](http://dx.doi.org/10.15242/IIE.E0514530)

Fig. 1 Graphical symbols used to represent the different types of vertices and edges in ESDG.

IV. 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 and
- Many field of software engineering.

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:
- 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. Each individual in the selection pool receives a reproduction probability depending on its 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 the population.
- 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.
- Mutation: Mutation operation alters chromosomes in small ways to introduce new good traits.

V. REGRESSION TEST FRAMEWORK

This paper presents an approach for the selection of test cases $T'$ from the 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 4 illustrate the various activities of the test case selection framework.

- Test coverage generation <CoveraInfo>
- ESDG Constructor <ESDG Mode>
- Identify changes <changed>
- ESDG Updates <M'>
- Test case selection <T'>
- Affected statements Identification <affectedStatements>
- GA based Prioritization <GAPriorTCase>

The scopes of the changes in our approach are addition and deletion of object.

1) Adding object:

Adding object in ESDG can be identified by Identify changes phase. Adding of object in object-oriented programming can be addition of method call statements, or simple statements such as conditionals, loops and assignment statements in the program.

Fig 3a and fig 3b (i, ii) represent program P and its modified version $P'$ codes, and their ESDGs of addition of method call statement.

In fig 3a, method call statement $\text{sum}(y, 1)$ was added in line S6a. Fig 3bii represents the method call statement added in the code in line S6a.

```
CE1 class A {
  E2 public int x, y;
  E3 void A () {
    S4 x = 5;
    S5 y = 7;  }
  E6 void increment(y) {
    S6a sum(y, 1); // added method call statement
  }
  E7 void sum(int x, int y) {
    S8 x = y;
  } // class
}
```

Fig. 3a Addition of method call statement in the code

2) Deleting of object:

An example of deletion of simple statement has been presented in fig 4a, and in the case of deletion of method call statement, we will used the code and ESDG in fig 3a and b respectively. In fig 4aii, the deleted node is S4 marked by dash line.

```
CE1 class A {
  E2 public int x, y;
  E3 void A () {
    S4 x = 5;
    S5 y = 7;  }
  E6 void increment(y) {
    S6a sum(y, 1); // added method call statement
  }
  E7 void sum(int x, int y) {
    S8 x = y;
  } // class
}
```

Fig 3b(i and ii). updated ESDG model for addition of method call statement

In fig 3a, we assume the deleted method call statement is S6a. To update the model by deleting the node S6a in ESDG, first identify the changed nodes, i.e., nodes that are control dependent or data dependent or dependent due to object relation such as inheritance and polymorphism, and saved these nodes in the file named “changed” to be used later. Secondly, there is need to remove all the parameter edges, the
Assuming the affected nodes are: n1 n2 n3 n5 n6, identified by performing forward slicing using “changed nodes” as slice criterion.

The selected test cases will be:

\[ t1 = \{n1 n2 n3 n6\}, \quad t2 = \{n1 n2 n3\}, \quad t3 = \{n1 n2\}, \quad t4 = \{n1 n2 n5 n6\}, \quad t7 = \{n1 n2 n6\}, \quad t8 = \{n1\}, \quad t9 = \{n1 n2 n3 n4 n5 n6\} \]

I.e., the selected test case \(T'\) will be:

\[ T' = \{t1 t2 t3 t4 t7 t8 t9\} \]

G. Test Case Prioritization

The selected test cases need to be ordered to increase the rate at which faults are detected while running them. The ordering can be done randomly, but most of the test cases randomly ordered first may result in checking small parts of the modified software. Test case prioritization technique needs to be given some guides in order to improve it efficiency. We proposed an evolutionary approach using GA to optimize the selected test cases. The selected test cases \(T'\) will be prioritized using evolutionary search algorithm and denoted as “\(GA\)Prior\(T\)Case”.

Example1.

Assuming we have the following test cases as selected in the step above.

I.e., \(T' = \{t1 t2 t3 t4 t7 t8 t9\}\)

The selected test cases \(T'\) will be prioritized using genetic algorithm.

Given problem:

\[ T1 = 4 nodes, \quad T2 = 3 nodes, \quad T3 = 2 nodes, \quad T4 = 4 nodes, \quad T7 = 3 nodes, \quad T8 = 1 node, \quad T9 = 6 nodes. \]

Solution:

1. Encode the solution

\[ T' = \{1 2 3 4 7 8 9\} \]

2. We can create n population randomly or reverse the order to have second parent/chromosomes.

\[
\begin{align*}
P1 & = \begin{bmatrix} 1 & 2 & 3 & 4 & 7 & 8 & 9 \end{bmatrix} \\
P2 & = \begin{bmatrix} 9 & 8 & 7 & 4 & 3 & 2 & 1 \end{bmatrix}
\end{align*}
\]

Our generated population (2 chromosomes) will serve as the initial population.

3. Evaluate the population by computing the fitness of each chromosome/parent.

\[
Ft(pi) = \sum_{j=1}^{n} n(t_j) \cdot P_{ij}
\]

Where \(P_{ij}\) is the position of the test case \(j\) in the parent/chromosome \(i\), and \(n(t_j)\) is the number of affected nodes in the test case \(j\).

4. Perform crossover operation on the two parents

\[
\begin{align*}
P1 & = \begin{bmatrix} 1 & 2 & 3 & 4 & 7 & 8 & 9 \end{bmatrix} \\
P2 & = \begin{bmatrix} 9 & 8 & 7 & 4 & 3 & 2 & 1 \end{bmatrix}
\end{align*}
\]
Generate a random number between 1 and 6 to be used as crossover point. A valid solution would needs to represent a child where every test case is included at least once and only once. We will use ordered crossover to produce valid child for next generation. A subset of the first parent will be selected and added to the child. Genes/test cases which are not yet in our child are added from the second parent in their order. For example if the random number is 4, copying the first 4 genes from the first parent gives:

\[
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
\end{array}
\]

And copying the remaining from the second parent give us:

\[
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
9 & 8 & 7 & 6 \\
\end{array}
\]

5. Mutate the child
The mutation operation also needs to be adjusted, so that it will not add random test case to the child, possibly causing a duplicate. We will use swap mutation to avoid duplication of test cases. Two genes will be selected at random then we simply swap their position.

Suppose the two randomly generated numbers are 3 and 7, our child will be:

\[
\begin{array}{cccc}
1 & 2 & 7 & 9 \\
4 & 8 & 3 & 6 \\
\end{array}
\]

6. Evaluate the fitness of the child C1 and replace the worst chromosome from the population.

7. Check if the termination condition not true, repeat step 4 to 6 else end the process.

VI. CONCLUSION

An evolutionary prioritized test case approach is 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 changed information 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.

A tool will be developed based on our proposed framework to be used in object-oriented programs, and we will compare results from our developed tool to measure the preciseness, inclusiveness and rate of faults detection.

REFERENCES


http://dx.doi.org/10.1109/ICSM.1994.336793


http://dx.doi.org/10.1145/248333.248262


http://dx.doi.org/10.1002/smr.236


http://dx.doi.org/10.1002/1099-1689(200006)10:2<77::AID-STVR197>3.0.CO;2-E


http://dx.doi.org/10.1109/ICST.2010.64


http://dx.doi.org/10.1145/77606.77608


http://dx.doi.org/10.1109/ICSE.1996.493444


http://dx.doi.org/10.1007/s40012-013-0011-7


http://dx.doi.org/10.1007/s11334-013-0221-z


http://dx.doi.org/10.1109/TSE.2007.38

http://dx.doi.org/10.15242/IIE.E0514530