Hindawi Publishing Corporation
Advances in Software Engineering
Volume 2009, Article ID 535708, 10 pages
doi:10.1155/2009/535708

Research Article

Regression Test Selection for C# Programs

Nashat Mansour and Wael Statieh

Department of Computer Science and Mathematics, Lebanese American University, 1102 2801 Beirut, Lebanon

Correspondence should be addressed to Nashat Mansour, nmansour@lau.edu.lb

Received 5 August 2008; Revised 4 January 2009; Accepted 23 May 2009

Recommended by Mauro Pezzè

We present a regression test selection technique for C# programs. C# is fairly new and is often used within the Microsoft .Net framework to give programmers a solid base to develop a variety of applications. Regression testing is done after modifying a program. Regression test selection refers to selecting a suitable subset of test cases from the original test suite in order to be rerun. It aims to provide confidence that the modifications are correct and did not affect other unmodified parts of the program. The regression test selection technique presented in this paper accounts for C#.Net specific features. Our technique is based on three phases; the first phase builds an Affected Class Diagram consisting of classes that are affected by the change in the source code. The second phase builds a C# Interclass Graph (CIG) from the affected class diagram based on C# specific features. In this phase, we reduce the number of selected test cases. The third phase involves further reduction and a new metric for assigning weights to test cases for prioritizing the selected test cases. We have empirically validated the proposed technique by using case studies. The empirical results show the usefulness of the proposed regression testing technique for C#.Net programs.

1. Introduction

Testing software is an important part of the production life cycle of a program. Testing is an expensive activity. Hence, appropriate testing methods are necessary for ensuring the reliability of a program. Regression testing aims to provide confidence in the correctness of a program after its modification. During the initial development of the program, a set $T = \{t_1, t_2, \ldots, t_N\}$ of N test cases is saved and a table of test case-method coverage information can be determined. After a program is modified, regression test selection requires a subset of test cases, R, to be selected from T for rerunning on the modified program with the objective of providing confidence that no unintended effects have been caused by the modification.

It would be costly for regression testing to repeat the whole set of test cases T used in the initial development of the program and unreliable to choose a random subset of test cases from T. Therefore, it is necessary to select a suitable subset of test cases from T to run. Regression test selection reduces the cost of testing by focusing on the changes that take place in the program.

A number of regression test selection approaches have been developed. Several approaches and techniques have addressed the problem of regression testing for procedural programs. Examples of procedural-based techniques are: slicing, data flow, firewall, and optimization [1–4]. Others have dealt with object oriented (OO) programs. Some of these OO techniques for regression test selection are based on UML diagrams and make use of only design information. Examples of these techniques are presented in [5–9]. Also, object oriented firewall techniques have been proposed in [10]. Extended firewalls are proposed for Object Oriented Regression Testing in [11]. Their algorithm makes a distinction between affected components and checked components. Then, it detects all the components that directly or indirectly call the modified component. These are called faults. The authors find that the extended firewall revealed more faults than normal firewall techniques. The object oriented code-based techniques have been based on Java and C++ to handle the interprocedural control flow and the features of these languages. To handle the Java constructs and features, a regression test selection algorithm has been developed which builds a Java interclass graph as an extension of the control flow graph [12]. Rothermel et al. [13] have addressed the

regression test selection technique problem for C++ software which is a code-based technique that builds an interclass control flow graph in order to find the difference between the original and the new programs. Jang et al. [14] presented another regression testing technique for C++ software. The authors mainly focus on functions that should be retested in C++ programs. They identify the type of change, and a firewall of its type is determined in order to find the dependency between the statements. A unit firewall is a set of member functions which require unit level retesting; an integration firewall is a set of interactions between member functions which require integration level retesting. Recently, Li and Harrold [15] presented a method for random test selection based on the Chernoff bound. Chittimalli and Harrold [16] described a regression test selection technique based on system requirements. Also, Qu et al. [17] proposed configuration-aware regression testing technique for evolving software systems using combinatorial interaction testing, configuration sampling, and prioritization.

C# is a fairly new object oriented programming language which is integrated in the Microsoft's .NET framework for application development. However, no work has been reported on regression testing C# applications with their specific features. The objective of this paper is to present, for the first time, a code-based regression test selection technique for C# programs. This technique is a three-phase technique. In the first phase, we build an Affected Class diagram covering the classes that are changed or affected by such changes in the source code. In the second phase, we develop a C# interclass graph, which represents the control flow of the methods in the classes considered in the affected class diagram and their interrelationships. This phase extends previous object oriented techniques by covering C# specific features. In the third phase, we assign weights to test cases using a proposed metric. According to the weights, we reorder the test cases that need to be rerun. We show the usefulness of our technique by presenting the results of applying it to a few examples.

This paper is organized as follows. Section 2 presents some assumptions and describes required data structures. Sections 3–5 describe our technique for selecting test cases from the initial suite of tests. Section 6 presents and discusses our empirical results. Section 7 proposes further work. Section 8 concludes the paper.

2. Assumptions and Notation

We assume that we have the test suite, T, consisting of the test cases determined during the initial development of the program. Each test case in T covers one or more methods in the classes. We use an adjacency table to represent the test case-method coverage information. After modifying the program, we also save the set of directly changed methods, M_1, and the set of deleted methods, M_2.

The set of affected methods are the methods that are directly or indirectly calling the changed, added, or deleted methods. These methods are gathered from the trace file produced by the .Net framework. The main purpose of the trace file is for testing and optimization after an application is compiled and released. We instrument the trace file by writing information about each method that calls another method (e.g., $A.m$ calls $B.n$, where A and B are classes; m and n are methods).

Table 1: Notation used.

Notation	Description
CD	Class Diagram
C	Set of classes of the original program
C'	Set of classes in the modified program
CIG	C# Interclass Graph
M_1	Set of directly changed methods
M_2	Set of deleted methods of the program
M_3	Set of methods that explicitly call the changed, and deleted methods
M	$= \{M_1 + M_2 + M_3\}$
AC	Class that contain a changed or deleted method
ACD	Affected Class Diagram that contains classes affected by modification in source code.
T	Test cases saved in the initial development of the program
AT	Test-method coverage table for the methods and their corresponding test cases
T_i	Test case that covers method m_i
T'	Set of test cases from the test suite that needs to be retested that cover classes in ACD
T''	Set of test cases that traverse the affected or potentially affected edges ED
ED	Set of affected or potentially affected edges that reach modified code
T'''	Reduced test cases from T''
TF	Trace file generated by the C#.net program

Table 1 lists the notation used in the rest of the paper.

3. Technique Overview

Our technique for test case selection consists of three phases.

Phase-1. Test case selection based on the affected class diagram (ACD).

Step 1. Generate the ACD that contains: all the classes that contain modified and/or deleted methods; the base classes and derived classes of the changed classes; the classes that use the changed classes (containing methods that call directly or indirectly the modified methods). In addition to classes, the ACD may also contain the following: Interfaces; Web and windows services, which is the use of an external method or class outside the program on a server or on the internet; COM+ components which are a serviced component registered by .Net and deployed on a server so that many applications can use it. The advantage of using COM+ components is its reusability.

Step 2. This step involves selecting $T' \subseteq T$ based on the ACD. We use the test-method coverage table and the test suite T for finding T' that cover ACD.

Phase-2. Test case selection based on the C# Interclass Graph (CIG).

Step 3. Generate the CIG which is a control flow graph that involves all the affected methods of the classes in ACD. The graph takes as input the set of classes in ACD and builds the control flow graphs of their affected methods.

Step 4. This step involves selecting a set $T'' \subseteq T'$ based on the CIG. Given the CIG graph of the old and the modified programs, we traverse the same paths in the two CIG Graphs by traversing the edges until we reach a difference in the target nodes. This edge will be marked as affected or potentially affected edge and will be saved. Any test case that covers this edge will be selected for rerunning.

We traverse all the paths of the two graphs until we find all the set of affected or potentially affected edges and select the test cases $T'' \subseteq T'$ that cover them.

Phase-3. Further reduction and prioritization.

Step 5. This step involves reducing the number of test cases selected from the first or second phase. In the first phase, for each affected method, we randomly select one test case covering this method. In the second phase, we randomly select one test case covering each affected or potentially affected edge.

Step 6. Prioritize test cases in T' or T'' based on the tester's choice whether to go for the two phases or stop at the first phase.

The six steps are explained in detail in the following sections.

4. Test Selection Based on ACD

We build an Affected Class Diagram that includes classes that contain modified or deleted methods. Then, the supertypes of these classes are determined and added to the Affected Class Diagram. We continue finding the derived classes for all the classes in ACD. But, we select only classes that contain a method overriding a modified/deleted method. The edge notations used in ACD are the following.

(i) Inheritance edge: is a supertype edge going from the derived class to the base class.

(ii) Use edge: is used when a class contains a method that explicitly calls another class.

(iii) Indirect subtype edge: is used when a class contains an overridden method and a class contains a method overriding that method. Their will be an indirect subtype edge between the two classes.

Algorithm 1 presents the algorithm used for building an Affected Class Diagram. The ACD includes the classes that need to be, next, expanded in the C# interclass graph.

Test case selection in this phase involves selecting the set of test cases T' that cover affected methods of the classes in ACD. Algorithm 2 shows the algorithm for selecting T'.

5. Test Selection Based on CIG

5.1. Building the C# Interclass Graph. The C# interclass graph (CIG) is based on the Affected Class Diagram. This graph represents the control flow between the different methods of the classes in the ACD. In order to select the test cases that need to be rerun, we have to build two CIG graphs for the original and modified programs. We begin traversing the same edges of the two programs until we detect a different target node of the same edge. We mark this edge as affected or potentially affected and add it to the set of affected or potentially affected edges. After traversing the whole CIG graph, we obtain a set of edges marked as affected or potentially affected. From these edges, we select the test cases $T'' \subseteq T'$ that cover them. In the following subsections, we describe the CIG.

5.1.1. Edge Notation for the CIG Graph. The C# interclass graph uses a set of edges to identify the type of method calls and the control flow between statements inside the method. A control flow edge represents the flow of consecutive statements inside a method. The call edge represents an explicit call to a method. The external path edge is an edge that identifies a call to an external component such as a class library or a COM+ component. This edge denotes the existence of an external component that the application uses to perform certain functionality. The path edge represents or summarizes the paths through the method called.

Each edge in the C# Interclass Graph has a label with the type of receiver instance that causes the method to be bound to the call. The edge notation for the CIG graph is represented as follows:

Control Flow edge ⟶
(Method) Call edge ⋯⋯→
External Path Edge - - - -→
Path edge ⟶

5.1.2. Nodes Represented in the CIG Graph. Class library: is a set of compiled classes you use inside your program. They are not expanded because it is a reusable component. The C# program instantiates an object of that class and uses its methods. We represent the call to a Class library by a dotted circle node and an exit node. There is a path edge between them which represents the path inside the class library. We encode the name of the class library, the class and method called in the class library node. If we changed the called method inside the class library, all nodes calling this class library will certainly be detected. The notation used is represented in Figure 1.

Calling a Web Service. We call these services to do a certain calculation and return a value that is of use to the C# program. These are classes on the internet that do specific functionalities. We represent the call to a web service by a call and a return node. We encode the web service name and the method call inside the node. This allows us to detect the classes that call the web service. The notation is represented in Figure 2.

Delegates: are like old function pointers, they refer to some methods based on the type of objects it instantiates.

Input: Set of methods $M = \{M_1 + M_2\}$ inside the classes;
Set of classes (C) of the whole program;
AC is an affected class that contains a changed/deleted method;
CD is the class diagram for the whole program;
ACD contains classes that have changed/deleted methods;
Output: set of classes that need to be considered for regression testing;
an Affected Class Diagram (ACD) with the relationship between the classes.
Description: this algorithm is used to select a subset from the whole classes of the program that are affected by the changes made to the program.

```
Begin
M3 = ϕ
   For each (AC) do
    {
        Find Base class (b) for AC from CD // where b is the super type of AC
        If (b) not in ACD
                {
                 Add (b) to ACD
                 Add an inheritance edge between AC and b them
                }
    }
- - up to this stage we have a set of changed classes with their base classes in ACD
with the relationship between them.
   For each class AC do
    {
         Get methods of AC
         if AC contains an overridden method m
    (for each class (d) in CD that contains a method n that overrides m) do
                {
                    if (d) not in ACD
                     {
                       Add (d) to the ACD Diagram
                       Add indirect subtype edge between (d) and (AC)
                     }
                }
    }
End
- - - Now to find the classes that explicitly reference the changed classes directly or
indirectly use the Trace file.
   Begin
      Open Trace file
         While not EOF (Trace) do
            Read Trace statement
               If (A.m calls B.n) && (B ∈ ACD) && (n ∈ M)
                  {
- -comment: A belongs to C and B in ACD and the method n in B belongs to M.
                     If (A) not in ACD
                        {
                         Add A to ACD
                         Add a use edge between A and B
                         M3 =  M3 ∪ n
                        }
                  }
          End While
M = M ∪ M3
End
```

Algorithm 1: Algorithm for building the affected class diagram.

Input: Affected Class Diagram (ACD);
AT which is a test-method coverage table for the methods and their corresponding test cases;
M: set of changed/deleted, and methods explicitly calling them.
Output: Set of Test cases T' that cover M from the original test suite
Description: This algorithm selects the test cases T' that cover the affected methods by modification in the program.

```
T' = φ
Begin
  For each Class.method ∈ (AT)
    {
      If Class ∈ ACD && method ∈ M
        {
          For each test case t_i that covers m
          T' = T' ∪ t_i
        }
    }
Return T'
End
```

Algorithm 2: Algorithm to generate subset of test cases T'.

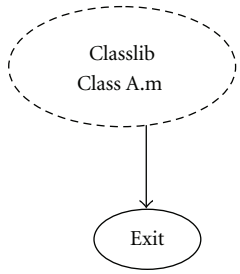

Figure 1

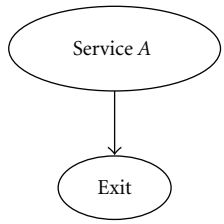

Figure 2

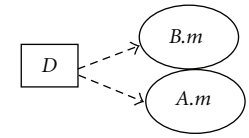

Figure 3

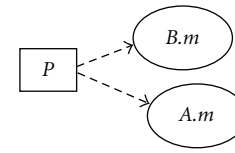

Figure 4

Delegates can be polymorphic which allows the delegate to be bound to many methods. The notation is represented in Figure 3.

Pointers: are represented in Figure 4.

COM+ Components. They are not changed; they are treated as black boxes. The COM+ components are registered class libraries found on the server and can be called from the C# program. Also the component might be on the internet. We encode the name of the component, the class, and the method called. The notation is represented in Figure 5.

Call to Read from XML File. There might be interaction between C# programs and XML files. The call to read from an XML file is treated the same as a call to an internal method. We encode the name of the XML file and the path of this file inside the node. This allows us to detect the classes that call XML files. The notation is represented in Figure 6.

Call to Execute a Stored Procedure. C# programs interact with stored procedure by sending the name of the stored procedure to the SQL server inorder to execute it. These procedures run certain queries on the database and return results to the program. We represent calls to stored procedures by a call and a return node as any call to an internal method. The node will contain information about the name and location of the stored procedure. The notation is represented in Figure 7.

Events. Events in C# are handled like a call to a method. Each event is handled by an event handler. You raise events by supplying the address of the handler; a method that will receive the event. A call to a handler is similar to a call to any internal method in the program. We encode the name of the method called inside the node. Each call to a handler is represented by a call and a return node. The notation is represented in Figure 8.

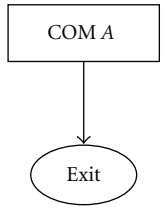

Figure 5

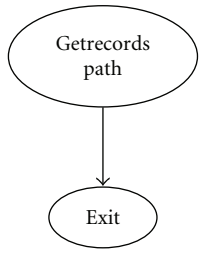

Figure 6

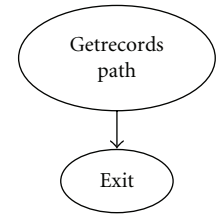

Figure 7

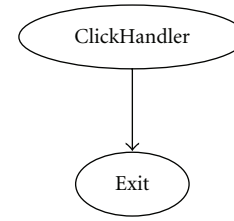

Figure 8

Events with Multiple Handlers. Events might be handled by more than one handler; all methods handling that event are executed. The node of the event raised will be associated with all the handlers, if event x is associated in code with handlers y, z, and q. When the event is raised, the order of handler execution will be y, z, and q. The notation is represented in Figure 9.

Exception Handling. When a C# application encounters a run-time error, it raises an exception and the application terminates. C# allows structured exception handling, which is a way for your application to recover from unexpected error or at least to have a chance to save data before closing. C#.Net contains built-in exception handlers to provide us with information about the exception. Also, the programmer can create catch blocks to catch the exception raised. Exceptions are treated like any other statements in the program.

5.1.3. Assumptions for the CIG Graph. (i) A statement change is a modification. Modification or deletion of an executable statement in a method is also a change. This is easily handled by selecting test cases that traverse them.

(ii) Modification of a variable or a constant in C# can be handled easily because these constants are put in an enumeration method and each method calling this enumeration will be affected.

(iii) Use a globally qualified name for each class. The globally qualified name lists all the hierarchy of such a class. Write the hierarchy information at each point of instantiation.

(iv) Each call to an external method (Class Library, COM+ component, Stored procedures, XML file, Web service) is represented as a pair of call node and a return node. They are connected with an edge called a path edge.

(v) Each call to an internal method is represented as a pair of call and return node.

5.2. Test Case Selection. Algorithm 3 presents the algorithm of finding the affected or potentially affected edges as a result of comparing the initial CIG graph with the modified one. The algorithm Compare is invoked at the entry node of each method that belongs to the classes of ACD. It accepts a node as input from the CIG and finds the affected or potentially affected edges; edges that cover or reach modified code. Algorithm 4 presents the algorithm for determining the set of test cases T'' that cover the affected or potentially affected edges.

Table 2: Characteristics of all subject programs.

Subject program	Classes	Methods	Relationships
Purchase order application-1	10	61	10
Purchase order application-2	10	61	10
Task management application-1	8	60	9
Task Management application-2	8	60	9
Archive application-1	14 1 class library	62	24
Archive application-2	14 1 class library	62	24

5.3. Step 5—Test Cases Reduction. The first phase of the regression test selection technique selects $T' \subseteq T$ that cover modified methods. In the first phase, T' is based on searching for the deleted/modified and affected methods and on selecting all the test cases that cover them. In the second phase, T'' ($\subseteq T'$) is based on selecting all the test cases that cover the affected or potentially affected edges. Instead of determining T' (or T'') based on selecting all test cases traversing an affected method (or affected/potentially affected edge), we can reduce the size of T' or T'' by randomly choosing one of these test cases. We refer to the resulting set of tests as T''', being a subset of T' or of T''.

```
Algorithm: Compare
Input: N: the entry node of the CIG of the original program P
       N′: the entry node of the CIG of the modified program P
Output: ED, which is the set of affected or potentially affected edges
Begin
   Mark N as visited
   For each edge e′ leaving N′ do
      e = match(N, e′)
      If e = null then continue
      C = get-target (e)
      C′ = get-target (e′)
      IF C <> C′ then
         ED = ED ∪ e
      Else
         If C is not marked as visited
            Compare (C, C′)
         End if
   For each edge leaving N and not found in N′ do
      ED = ED ∪ e
   End For
End compare
```

Algorithm 3: Algorithm to find affected or potentially affected edges in the CIG graph.

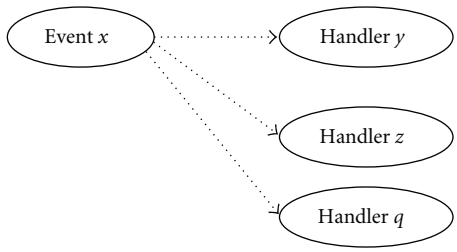

Figure 9

This reduction may be justified in the case of large T' or T'' for the purpose of reducing regression testing time.

6. Empirical Results

6.1. Empirical Procedure. The Regression test selection technique is applied to three applications: an Archive program, a Purchase order application for selling inventory, and a Task Management application that handles the internal tasks of a software company. For each application, we use different versions that lead to a total of six subject programs. Table 2 presents the characteristics of these subject programs. Initial information and data structures are generated for each version including the original test suite T, a test-method coverage table (AT), and a trace file. Each program undergoes logical change. Then, the three phase technique is applied to each subject program. We assess the regression test selection technique according to the following three criteria.

(i) The number of test cases selected by the technique denoted by T' or T''.

(ii) Inclusiveness determines the degree by which the technique selects modification revealing test cases [18]. Modification revealing tests are those that produce different output than the original program. mr is the number of modification revealing tests. When a regression test selection method selects some of these tests (denoted by smr), inclusiveness is given by 100(smr/mr).

(iii) Precision determines how the technique avoids choosing test cases that would not produce different output than the original program [18]. The non-modification revealing tests are denoted by nmr. The test cases that are non-modification revealing are omitted (denoted by onmr). Precision is given by 100(onmr/nmr).

6.2. Results and Discussion. Table 3 presents a summary of the results for the six subject programs. N is the total number of test cases (in the initial suite); N' is the number (and percentage) of selected test cases after the first phase; N'' is the number (and percentage) of selected test cases after the second phase. Tables 4 and 5 illustrate the precision and inclusiveness for the six examples after applying the first and second phases, respectively. Table 6 presents N''', the number of tests selected after applying the further reduction step (Step 5) on T''.

The number of selected test cases is considered relatively small (Table 3) ranging from 4.3% to 45% for N' (after applying the first phase) and 1.5% to 35% for N'' (after applying the second phase). This shows that our technique does lead to a reduction in the number of test cases to be rerun after program modification.

Input: Set ED of affected or potentially affected edges traversed in the CIG graph
Set of test cases T' that covers the classes in ACD
Output: set T'' of test cases selected from T'.
Description: this algorithm is responsible for finding the test cases T'' from T' that traverse those affected or potentially affected edges. Selection is based on the test cases selected from the affected class diagram.

```
T'' = φ
  ∀e ∈ ED do
    {
      For each t_i ∈ T'
        {
          If t_i covers e    // if the test case t_i traverses the modified edge e
          T''=T' ∪ t_i
        }
    }
Return T''
```

Algorithm 4: Algorithm to find T''.

Table 3: Number of selected test cases.

Subject program	N	N'	N''
Purchase application-1	200	90 (45%)	55 (35%)
Purchase application-2	200	45 (22.5)	5 (2.5%)
Task management-1	423	39 (9.2%)	23 (5.4%)
Task management-2	423	20 (5%)	6 (1.5%)
Archive application-1	460	27 (6%)	16 (3.5%)
Archive application-2	460	20 (4.3%)	8 (3%)

Table 4: Precision and inclusiveness results after the first phase.

Subject program	N	N'	Inclusiveness %	Precision %
Purchase application-1	200	90	100	71
Purchase application-2	200	45	100	79
Task management-1	423	39	100	87
Task management-2	423	20	100	96
Archive-1	460	27	100	90
Archive-2	460	20	100	96

Table 5: Precision and inclusiveness results after the second phase.

Subject program	N	N''	Inclusiveness %	Precision %
Purchase application-1	200	55	100	94
Purchase application-2	200	5	100	100
Task management-1	423	23	100	100
Task management-2	423	6	100	99
Archive-1	460	16	100	99
Archive-2	460	8	100	99.5

Table 6: Precision and inclusiveness after further reduction (Step 5) on T''.

Subject program	N	N'''	Inclusiveness %	Precision %
Purchase application-1	200	11	24	100
Task management-1	423	4	18	100
Archive-1	460	4	25	100

Tables 4 and 5 depict the precision and inclusiveness outcome for the first two phases of the technique. The inclusiveness of T' is 100%, which implies that all modification revealing tests are selected. The precision of T' is fairly high and ranges between 71% and 96%, which implies that a limited number of nonmodification revealing tests are selected. Table 5 shows that the inclusiveness of T'' is still 100%. That is, Phase 2 of our technique is also safe in selecting all modification revealing tests. The precision of T'' is close to 100%, which means that hardly any nonmodification revealing tests are selected. We also note that the precision after Phase 2 (of T'') is higher than that of Phase 1 (of T') for the same inclusiveness level.

Table 6 presents the results of the further reduction step (Step 5 in Phase 3), which show that the inclusiveness of T''' has dropped (to lower than 26%) in comparison with 100% for T' and T''. This drop occurs due to the omission of all redundant and multiple tests that cover the same elements. Hence, Step 5 of our technique does not provide safe coverage. However, the precision rises to 100% and all nonmodification revealing tests are omitted. We propose using this step as well as the prioritization step of Phase 3 only where N' or N'' is large and we have very limited time for regression testing; this would apply in emergency situations where more regression testing time is too expensive vis-à-vis the suspended services of a system

(waiting for the completion of regression testing). Also, the set T' of test cases chosen in the first phase will cover the affected or potentially affected edges in the ACD. Thus, building CIG graph, in the second phase, will result in selecting a set $T'' \subseteq T'$ to be rerun. Testers might not have time to go for two phases. They can stop at the Affected Class Diagram level and select and run all the test cases T'. This depends on the following condition:

$$T_{\text{ACD}} + T_{\text{select } T'} + T_{\text{run } T'} < T_{\text{ACD}} + T_{\text{select } T'} + T_{\text{CIG}} + T_{\text{select } T''} + T_{\text{run } T''}, \quad (1)$$

where T_G is the time to build the graph G, $T_{\text{select } X}$ is the time to select subset of tests X, and $T_{\text{run } T}$ is the time to run the subsets of tests X.

Our approach has a limitation when it is applied to large programs. The complexity of the algorithms is quadratic in the number of CIG nodes. For large CIGs, our technique can be costly. Hence, we need to explore ways to reduce the cost and still maintain effective regression testing.

7. Further Work—Prioritizing Test Cases

Test case prioritization entails running test cases with the highest priority earlier. Prioritization techniques can provide earlier detection of faults in the modified program. Test cases can be prioritized according to different metrics. They can be prioritized according to the number of covered methods or statements. However, prioritizing test cases according to the number of statements might end up running, first, test cases that cover large number of statements, but small quantity of modified code. A similar argument applies for methods.

Since our test selection techniques are based on safe coverage of all affected or potentially affected edges, they might lead to selecting a large number of test cases. Hence, prioritizing the selected test cases may be useful in further reduction of test cases. We propose a new prioritization technique that employs the number of modified methods. The weight of each test case relies on how much this test case cover modified or affected methods.

We treat class library changes differently from nonclass library changes. Since library classes are reusable components by many applications, it is important to assign the highest priority to the test cases that cover library class changes. Therefore, for a test case, T_i, that cover a library class, we assign maximum weight; that is, $W(T_i) = 1.0$.

For nonlibrary classes, we propose another expression to find the priority weight of a test case, T_i, as follows:

$$W(T_i) = NW * \frac{C_{NW}}{MT}, \quad (2)$$

where NW is the number of modified/deleted and affected methods covered by T_i, C_{NW} is the number of classes that cover the NW methods, and MT is the total number of methods in the program. The priority weight is proportional to both the diversity of affected methods and classes, normalized with respect to the total number of methods. That is, a selected test case is more important if it covers more methods distributed among more classes. Test cases are ordered according to the weight $W(T_i)$. This helps in selecting more important test cases first.

8. Conclusion

We have presented a regression test selection technique that is based on identifying changes in programs. This technique covers C# specific features and includes relevant .Net Features. The technique is able to select suitable subsets of test cases that are modification revealing. These subsets show a reduction in the number of tests selected for adequate levels of inclusiveness and precision. This has been demonstrated by the empirical results. Despite the limited number and size of the subject programs, the results provide confidence in the effectiveness of our technique. However, our technique also provides the tester with the flexibility to use phases and steps that are compatible with the required level of inclusiveness and precision, and allowed regression testing time.

References

[1] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. A. London, "Incremental regression testing," in *Proceedings of the International Conference on Software Maintenance (ICSM '93)*, pp. 348–357, Montreal, Canada, September 1993.

[2] R. Gupta, M. J. Harrold, and M. L. Soffa, "Program slicing-based regression testing techniques," *Software Testing Verification and Reliability*, vol. 6, no. 2, pp. 83–111, 1996.

[3] H. K. N. Leung and L. White, "A firewall concept for both control-flow and data-flow in regression integration testing," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '92)*, pp. 262–271, Orlando, Fla, USA, November 1992.

[4] N. Mansour and K. El-Fakih, "Simulated annealing and genetic algorithms for optimal regression testing," *Journal of Software Maintenance: Research and Practice*, vol. 11, no. 1, pp. 19–34, 1999.

[5] L. C. Briand, Y. Labiche, K. Buist, and G. Soccar, "Automating impact analysis and regression test selection based on UML designs," in *Proceedings of the 18th IEEE International Conference on Software Maintenance (ICSM '02)*, pp. 252–261, Montreal, Canada, October 2002.

[6] Y. Le Traon, T. Jéron, J.-M. Jézéquel, and P. Morel, "Efficient object-oriented integration and regression testing," *IEEE Transactions on Reliability*, vol. 49, no. 1, pp. 12–25, 2000.

[7] Y. Wu and J. Offutt, "Maintaining evolving component based software with UML," in *Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR '03)*, pp. 133–142, Benevento, Italy, March 2003.

[8] Q.-U.-A. Farooq, M. Z. Z. Iqbal, Z. I. Malik, and A. Nadeem, "An approach for selective state machine based regression testing," in *Proceedings of the 3rd International Workshop on Advances in Model-Based Testing (AMOST '07)*, pp. 44–52, London, UK, 2007.

[9] N. Mansour and H. Takkoush, "UML based regression testing technique for OO software," in *Proceedings of the IASTED International Conference on Software Engineering and Applications*, pp. 96–101, Boston, Mass, USA, November 2007.

[10] P. Hsia, X. Li, D. C. Kung, et al., "A technique for the selective revalidation of OO software," *Journal of Software Maintenance: Research and Practice*, vol. 9, no. 4, pp. 217–233, 1997.

[11] L. White, K. Jaber, and B. Robinson, "Utilization of extended firewall for object-oriented regression testing," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '05)*, pp. 695–698, Budapest, Hungary, September 2005.

[12] M. J. Harrold, J. A. Jones, T. Li, et al., "Regression test selection for Java software," in *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '01)*, pp. 312–326, Tampa Bay, Fla, USA, October 2001.

[13] G. Rothermel, M. J. Harrold, and J. Dedhia, "Regression test selection for C++ software," *Journal of Software Testing Verification and Reliability*, vol. 10, no. 2, pp. 77–109, 2000.

[14] Y. K. Jang, M. Munro, and Y. R. Kwon, "An improved method of selecting regression tests for C++ programs," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 13, no. 5, pp. 331–350, 2001.

[15] W. Li and M. J. Harrold, "Using random test selection to gain confidence in modified software," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '08)*, pp. 267–276, Beijing, China, October 2008.

[16] P. K. Chittimalli and M. J. Harrold, "Regression test selection on system requirements," in *Proceedings of the 1st India Software Engineering Conference*, pp. 87–96, Hyderabad, India, 2008.

[17] X. Qu, M. B. Cohen, and G. Rothermel, "Configuration-aware regression testing: an empirical study of sampling and prioritization," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '08)*, pp. 75–85, Seattle, Wash, USA, July 2008.

[18] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel, "An empirical study of regression test selection techniques," in *Proceedings of the International Conference on Software Engineering (ICSE '98)*, pp. 188–197, Kyoto, Japan, April 1998.