Regression Testing on Object-Oriented Programs

Ye Wu, Mei-Hwa Chen and Howard M. Kao
Computer Science Department
SUNY at Albany
Albany, NY12222
(wuye, mhc, kao)@cs.albany.edu

Abstract

Regression testing is an important activity at both testing and maintenance phases. When a piece of software is modified, it is necessary to ensure the quality of the software is preserved. To this end, regression testing is to retest the software using the test cases selected from the original test pool. In this paper, we present a regression testing technique that selects test cases by utilizing static information from the analysis of the program structure and dynamic information by tracing the function-calling sequences. To compare the effectiveness of this technique with other existing approaches, we conducted an empirical study on an industrial real-time system. The results show that not only does this technique preserve all the necessary information for regression testing, but it is also much more efficient and more precise than the existing techniques.

1 Introduction

Regression testing is aimed at ensuring that a modified software still meets the specifications that were validated prior to the modification activities. To this end, an intuitive approach would involve retesting the modified program using all the test cases in the test pool to make sure the program still preserves the quality that had been validated using these test cases. However, retesting all the test cases is often impractical due to development cost and delivery schedule constraints. A selective approach of regression testing is to choose a subset of the test pool that can provide sufficient confidence of the system. Therefore, a good algorithm that can be applied efficiently to select an effective regression test suite is strongly desirable.

A number of studies on selecting test cases in regression testing have been proposed. Most of them present techniques for traditional procedure based software [2, 3, 4, 7, 9, 10, 13, 15, 17, 19, 20, 21], but few address this issue in the object-oriented paradigm [1, 12, 16, 18]. Object-oriented programming has many useful features such as information hiding, inheritance, polymorphism, and dynamic binding. These features can be used to filter out test cases that are irrelevant to the modification, whereas they might require additional overhead in regression testing. For example, to take the runtime multiple instances into account, multiple copies of artifacts such as the control flow graph and/or program dependence graph might be necessary; to consider dynamic binding and polymorphism, dynamic analysis needs to be done. Therefore, the primary objective of our research on this subject is to develop a methodology that makes use of the properties of the object-oriented features to identify the portions of the program that were modified and those that were influenced by the modification. Yet, it does not require extra overhead to take these features into account.

A previous study on object-oriented testing [6] suggests that the function dependence relationships among the functions can be used to select a subset of inherited functions that must be retested in the derived classes. The results also show that such a subset of the inherited functions, as we selected, is sufficient, i.e., its effectiveness is equivalent to retesting every inherited function. Stemming from this study, we extend the utilization of the function dependence relationship in regression testing and present a selective regression testing approach for object-oriented programs.

Based on the dependence relationship among functions, when a modification is made to a program, we first identify which variables, functions and dependence relationships of the functions will be affected by the modification. A test case will be selected if and only if it will execute at least one of the modified functions that will influence the behavior of the program. Moreover, to investigate under what circumstance the program’s behavior would be influenced, we create a Function Calling Graph for the execution of each test case to trace the function execution sequence. With the static affected function dependence relationship and the dynamic function execution sequence information, this approach is able to select an effective regression test suite efficiently. Also, after comparing the affected and original de-
pendence relationships, we can provide the information of new occurrences of dependence relationships and the priorities of the dependence relationship. Based on this information, we can generate new test cases to test the modified program.

The paper is organized as follows: in the second section we describe the function dependence relationship. The selective regression testing approach which we proposed is detailed in Section 3, and the empirical studies conducted to demonstrate the strengths of this approach are presented in Section 4. We conclude our study and present our future research directions on this topic in Section 5.

2 Background

Function dependence relationship which is used in our previous work already shows its strength in OO testing. It focuses on a high level abstraction of the system, so it can be used efficiently. Meanwhile, the relationships among functions are based on the associations among data members and function call-return relationships. Thus it will still preserve the quality of the analysis. We will extend the concept of function dependence relationship to fit the needs of the regression testing. This section provides a brief description of the function dependence relationship.

A class defines the data relevant to an object of that class and a set of operations that may be performed on that data. Class data are referred to as instance variables or data members, and class functions are called methods or member functions. In this paper, we are more concerned with the data members than the local variables, so without the loss of generality, the variables we use in the paper are referred to as the data members within the same scope, so the effect of the local variables is transferred to that of data members. Moreover, the names of functions are considered as special data members as well.

Below we first describe relationships between functions and variables, followed by the definition of the function dependence relationship.

- A function $f$ uses a variable $x$ means that the value of the variable $x$ is referenced in an expression or is used to decide a predicate in $f$.

- A function defines a variable means that the value of the variable is assigned when the function is invoked.

- A variable $x$ directly uses a variable $y$ if one of the following conditions holds:
  1. The value of the variable $y$ is used in an expression to compute the value of the variable $x$.
  2. The value of the variable $y$ is used to compute the value of a parameter of a function and the return value of the function is used to compute the value of the variable $x$.
  3. The value of the variable $y$ is used in a predicate expression $e$, and a computation of the variable $x$ is located in one of the branches of $e$.
  4. The value of the variable $y$ is used to compute the value of a parameter of a function and the return value of the function is used in a predicate expression $e$. A computation of variable $x$ is located in one of the branches of $e$.

- A variable $x$ uses a variable $y$ means either:
  1. the variable $x$ directly uses the variable $y$, or
  2. the variable $x$ directly uses a variable $z$ and the variable $z$ uses the variable $y$.

The function dependence relationship is defined as follows: for functions $f_1$ and $f_2$, $f_1$ depends on $f_2$ if and only if one of the following conditions holds:

1. $f_1$ uses a variable $x$ that is defined in $f_2$.
2. $f_1$ calls $f_2$ and uses the return value of $f_2$.
3. $f_1$ is called by $f_2$ and the value of a parameter can be obtained from the value of a variable $x$ defined in $f_2$.

```
Class Shape {
  int perimeter, radius;
  public:
    Shape();
    double() { perimeter /= 2; }
    // In the modified version,
    // we will add statement radius/2;
    virtual int draw();
}
Class Square : Shape {
  int x, y;
  public:
    Square(int x1, int y1, int init_perimeter) {
      x = x1; y = y1;
      perimeter = init_perimeter;
    }
    void draw() { draw the square using x y and perimeter }
}
Class Circle : Shape {
  int x, y;
  public:
    Circle(int x1, int y1, int init_perimeter) {
      x = x1; y = y1;
      radius = init_perimeter;
    }
} 
```
perimeter = init_perimeter;
radius = init_perimeter/(2*3.14);
}
int draw() {draw the circle using x y and radius}
}
main()
{
    int size, perim, x, y;
    Shape shape_obj;
    While (1) {
        cin >> shape,perim,x,y;
        If (shape == 0) break;
        If (shape == 1)
            shape_obj=Square(perim, x, y);
        Else
            shape_obj=Circle(perim, x, y);
        shape_obj.draw();
        if (perim > 10) {
            shape_obj.double();
            shape_obj.draw();
        }
    }
}
test case 1: "1 12 2 2"
    - testing Square with perim 12

test case 2: "2 8 2 2"
    - testing Circle with perim 8

test case 3: "2 12 2 2"
    - testing Circle with perim 12

3 Methodology

To validate different regression testing approaches, Rothermel [17], Rosenblum [15], Graves [8] and Lung [14] provided different models to evaluate various regression techniques. Precision and efficiency [17] are the two major considerations in the evaluation. Precision is the measurement of the ability of a method to omit the test cases which do not need to be retested, while efficiency assesses the time and space requirements of the regression testing strategy. The objective of our regression testing technique is to efficiently and precisely select all the test cases which may cause the output of the modified program different from that of the original program. Intuitively, retesting all the test cases that execute the modified code is efficient and safe. Whereas, it is not necessary that every test case which executes the modified code will produce the output that is different from the original output. Therefore, the precision of the method can be improved. To this end, we may utilize the data-flow or dynamic slicing method to identify the portions of the program which will affect the output. However, such approaches often require large overhead that may reduce efficiency. In particular, in the OO paradigm, with OO features, such as inheritance, dynamic binding, polymorphism and message passing, the required overhead may be even less acceptable. By noticing that all of these OO features are related to the function calls which are associated with certain objects, we propose a regression testing technique based on the analysis of the dependence relationship among functions in the system. To preserve the precision provided by the data-flow method and to minimize required overhead, our method performs a two-phase analysis. The first phase is to analyze the affected variables, functions and function dependence relationships at statement level after the modification. The analysis will consider all possible effects of the modification on the system, so it is safe. Also, the analysis is performed at the statement level; thus, it will obtain a more precise function dependence relationship than the firewall and other methods. Meanwhile, this phase is a static analysis that only requires a one-time analysis; therefore, it is considerably efficient. To precisely process OO features and thus enhance the precision of our approach, in the second phase, by using the function calling graph (FCG) of each test case and the result obtained from the first phase, we can dynamically select test cases that need to be retested. To construct the FCG of a test case, we only need to record the calling sequence of functions, so the required overhead is proportional to the number of function calls.

In this section we first define what kinds of variables, functions and function dependence relationships are considered as affected and how to identify them. Then we present a simple test case selection algorithm by using the affected variables, functions and function dependence relationships.

3.1 Affected Function Dependence Relationship

The first step of our regression testing approach is to identify the affected variables, functions and function dependence relationships. Different from the relevant slicing method which considers the statement level dependent relations, and also different from the firewall wall method which only considers the relationships among modules either only based on the control-dependence or based on the inheritance, aggregation and association relationships. The dependence relationships we consider is at functions level but take data members into account.

In order to achieve this, we will first find the variables which will be affected by the modifications, furthermore we will identify functions and function dependence relationships which will be affected by the modifications based on affected variables.

directly affected variable:

A variable $x$ is considered as directly affected in function $f$ if either:
• a computation expression $ce$ which is used to compute the value of $x$ is modified in $f$, or
• a predicate expression $ps$ in $f$ is modified, and the computation expression of $x$ is located in one of the branches of $ps$, or
• $x$ is defined in $f$, and a function call to $f$ is added or deleted from the program.

affected variable:

A variable $x$ is considered as affected in function $f$ if either:

• $x$ is directly affected in $f$, or
• $x$ uses a variable $y$ and $y$ is directly affected in $f$, or
• $x$ is defined in $f$ and $f$ can be reached from $f'$, where either $f'$ appears in a branch of predicate statement $ps$ and $ps$ contains an affected variable $y$, or $f'$ is a function call being added or deleted from the program.

affected function:

A function $f$ is considered as affected if $f$ uses a variable $x$ and $x$ is affected in $f$.

The selection of test cases is based on the output of the program, if the output of the programs before and after the modifications are the same, we do not need to retest that test case. Therefore we more concern about the functions which may generate different output than other function which do not. To identify the functions which will produce different output, we will select statement which may generate different outputs first.

behavior-affected statement:

A statement is considered as behavior-affected if it is one of the following kinds:

• Output statements which are added to the program.
• Output statements which will output affected variables.
• Output statements which locate in one branch of a predicate, and the predicate contains an affected variable.
• While loop with an affected loop control part.
• Computation statements which may stop the normal execution when error occurs. For example computation expressions include divisions or square root and etc.

A statement which contains null pointer reference or array reference which may exceed the boundary.

behavior-affected function:

A function $f$ is considered as behavior-affected by a modification if $f$ is affected and contains behavior-affected statements or some output statements are removed from the function.

We can use behavior-affected function to filter out irrelevant test cases, but in many cases, even though a test case executes the behavior-affected functions, the output may not change. The reason is that we only execute the behavior-affected functions, but do not execute the modified/affected functions which cause the changes. To take care this kind of situation, we need to consider the behavior-affected functions as well as the modified/affected function, in other words, the affected function dependence relationship.

affected function dependence relationship:

A function dependence relationship between function $f_1$ and $f_2$ is considered affected if one of the following holds:

• $f_1$ uses a variable $x$ that is defined in $f_2$ and $x$ is affected in $f_2$.
• $f_1$ calls $f_2$ and uses the return value of $f_2$ and at least one of the variables which are used to compute the return value of $f_2$ is affected.
• $f_1$ is called by $f_2$ and the value of a parameter can be obtained from a variable $x$ defined in $f_2$ and $x$ is affected in $f_2$.

3.1.1 Affected Function Dependence Graph

An Affected Function Dependence Graph $G_f = (V_A, E_A)$ of a modified program is used to depict the affected function dependence relationships in the modified program. Where a node $v_i \in V_A$ represents a function in a class and a directed edge $e_{ij} = (v_i, v_j) \in E_A$ represents: function $v_j$ depends on function $v_i$ in the modified program, and the function dependence relationship between $v_i$ and $v_j$ is affected. If the affected function dependence relationship between $v_i$ and $v_j$ is based on the condition 1, then edge $(v_i, v_j)$ is a Sequence edge, otherwise, it will be a Call-return edge.

To construct an AFDG, we first classify all possible modifications into the following categories:

1. Add, modify or delete output statements.
2. Modify the definitions of some variables.
3. Add new definitions of some variables.
4. Remove definitions of some variables, which can be considered as modifying the definitions of those variables.

5. Modify, delete and add new uses of some variables in a computation expression, which can be considered as modifying the definitions of the variables computed by the computation expressions.

6. Modify, delete and add new uses of some variables in a predicate. Identify all the variables that are defined in all the branches and treat them as definition modified.

7. Add or delete function calls. First identify all the callees of these modified functions, if one of the callees contains behavior-affected statements, mark the modified function “behavior-affected,” otherwise, treat all variables defined in these functions as definition modified.

Described below is an simple iteration algorithm developed to identify all the affected variables, functions and function dependence relationships in a modified program, and create the Affected Function Dependence Graph of the modified program. The inputs of the algorithm include the program $P$, the function dependence graph and a modification history of $P$.

Algorithm1: Identifying the affected functions and function dependence relationships

Input: Program $P$, FDG and a modification history of $P$.
Output: AFDG

Step 1: Identify all affected variables:
Construct CFG for each of the functions in the system. For each modified function $f_i$, find out the variables which will be affected.
While (New affected variables are identified)
For each function $f_j$ in the system, search its CFG to find out new affected variables.

Step 2: Identify all functionality-affected functions:
For each function $f_i$ in the system
Search its CFG to find out whether it contains behaviour-affected statements.

Step 3: Identify all affected function dependence relationships:
Part 1: For each function $f_i$, search the CFG to find out its A-list, N-list and U-list; A-list contains all variables which are defined and affected in the function $f$. N-list contains all the variables which are defined in $f$, but will not be affected. U-list contains all the variables which are used in $f$.
Part 2: Add edges into the AFDG:

1. If ($f_i$’s A-list ∩ $f_j$’s U-list ≠ 0) Then add an sequence edge ($f_i$, $f_j$) into the AFDG.
2. If variable $x$ ∈ $f_i$’s A-list and $f_i$ calls $f_j$ where $x$ is used to compute one of the parameter, then add a call-return edge ($f_i$, $f_j$) into the AFDG.
3. If $f_i$ calls $f_j$ and uses the return value of $f_j$, when the return value of $f_j$ may change, add a call-return edge ($f_j$, $f_i$) into the AFDG.

In Figure 2 is shown the AFDG of the example described in Section 3 after the statement “radius = perim/2” is added to the program.

3.1.2 Analysis

In this section we present an analysis of the complexity of the algorithm described in Section 3.1.1. Below are the notations that will be used in the analysis of the algorithm:

$N_v$ = Total Number of data members in the system.
$N_{fc}$ = Lines of code in the system.
$N_f$ = Number of functions in the system.

The basic data structure used in the algorithm is control flow graph which can be easily obtained within $O(N_{fc})$ time.

The running time of step1 of the algorithm depends on number of iterations of the while loop. From the algorithm we can easily find out, the while loop will be continued if and only if there are new affected variables. There are at most $N_v + N_f$ different variables (includes the return value.
changes), so it will loop at most \(N_v + N_f\) times. In each iteration, we have to traverse the CFG which need \(O(N_{te})\) time. So the running time of step 1 is \(O((N_v + N_f) * N_{te})\).

In the steps of identifying behavior-affected statements, functions and function dependence relationships, we may traverse the program three times, once in step 2, once in searching for A-list, N-list and U-list, and once in identifying affected function dependence relationships. Additionally, we may need \(O(N_v + N_f^2)\) time to identify affected function relationship within one class.

Based on the above analysis, the algorithm can be done approximately within \(O((N_v + N_f) * N_{te})\) time. Identifying affected variables, functions and functions dependence relationship is the procedure need to be done once for the modified program. So it will not affect the efficiency of the whole regression technique.

### 3.1.3 Simple selection algorithm

By using the AFDG of the modified program we can apply a simple test case selection method. A test case will be selected only if the program will execute at least one modified function and one behavior-affected function using such test case. The running time of this simple test case selection algorithm is \(O(N_f)\).

To demonstrate that using the simple test case selection method in regression testing is safe, we first show that this method can identify all the affected variables, functions and function dependence relationships of a modified program, which can be concluded from the following observations:

1. If variable \(x\) is an affected variable, there must exist a sequence of sets of variables: \(V_1, V_2, ..., V_k\). Variables in \(V_i\) is affected because of the modification, and for any \(y\) in \(V_i\), \(y\) is affected because \(y\) uses some affected variable belong to \(V_{i-1}\). According to the step 1 of the algorithm, it is obvious that variables in the set \(V_i\) will be identified as affected after at most \(i\) iterations. Thus \(x\) will be identified as affected after at most \(k\) iterations.

2. Because we correctly identify all affected variables, after one more pass of the program, the step 2 can obtain all behavior-affected functions. Meanwhile, step 3 will traverse the program twice to acquire all function dependence relationships which are affected.

Let’s assume statements other than behavior-affected statements will not adversely affect the program. However, there are some kinds of situations needing special attention, for example: memory leak problem and so on. For these kinds of situations, different systems have different ways of handling them. Java itself already includes a memory management system, we do not need to consider this problem. Otherwise, we need to make all these kinds of statements “behavior-affected” statements as well.

Below we show why the simple selection method is safe.

1. If the test case does not execute any modified functions, with the same input, the codes that it will execute in the modified program are exactly the same as those it executed in the original version. There is no difference in the functionality of the two systems; therefore, we do not need to retest the test case.

2. If the test case does not execute any behaviour-affected functions, this indicates that all the statements it executes will not affect the output of the system dispite of the modifications. In other words, based on the assumption, for each test case, the outputs will be exactly the same as the original system. So we do not need to retest the test case.

### 3.2 Selection algorithm with FCG

We have shown that the simple test case selection method is efficient (Section 3.1.2) and proved that it is safe (Section 3.1.3.) However, the number of test cases selected using this method may not be precise for the following observations.

If function \(f_1\) depends on \(f_2\), and the function dependence relationship between \(f_1\) and \(f_2\) is affected. Such an affected function dependence relationship will not be in effect,

1. if \(f_2\) is not executed or if the execution of \(f_1\) is completed before \(f_2\) is executed,

2. if \(f_1\) and \(f_2\) may bind to different instances of a class, i.e., we need to distinguish different instances with which the function is associated.

3. if there are no function calls between \(f_1\) and \(f_2\), whereas the function dependence relationship between \(f_1\) and \(f_2\) is affected when there is a function call between \(f_1\) and \(f_2\).

4. if there are some function \(f\) which executed between \(f_1\) and \(f_2\), will restore the original dependence relationship of \(f_1\) and \(f_2\).

#### 3.2.1 Function Calling Graph

Simply recording the execution sequence of the functions works, but the size of the sequence is unbound. Also each instance has its own execution sequence, thus it is very expensive to determine the dependent relationships among functions. In this section we present the approach of using the Function Calling Graph and it’s advantages. In the next section, we will give a selection algorithm which makes use of the FCG.
A Function Calling Graph (FCG) is a Directed Graph $G_{FCG} = (V_{FCG}, E_{FCG})$. Each node $v_i \in V_{FCG}$ is a function in the program, and each $e_{ij} = (v_i, v_j) \in E_{FCG}$ represents two functions $v_i$ and $v_j$ where $v_j$ executes after $v_i$.

Function $v_i$ executes before $v_j$ iff one of the following holds:

1. If function $v_i$ calls function $v_j$ or $v_i$ is called by function $v_j$ (Call-Return edge in $G_{FCG}$).

2. $v_i$ and $v_j$ are functions associated with the same instance. In the execution history, there exists an execution of $v_i$ and $v_j$ satisfying the following: $v_i$ executes before $v_j$ and there is no other function, which is associated with the same instance, that executes between $v_i$ and $v_j$ (Sequence Edge in $G_{FCG}$).

3. $v_i$ and $v_j$ are functions in the same class. In the execution history, there exists an execution of $v_i$ and $v_j$ satisfying the following: $v_i$ executes before $v_j$ and there is no other function in the same class executed between $v_i$ and $v_j$ (Sequence Edge in $G_{FCG}$).

### 3.2.2 Selection Algorithm

Once we have the AFDG of the modified program and the FCG for each test case in the suite, the new selection algorithm will do the following. First, it identifies the variables which are really affected in this execution. Then, it determines whether a test case contains at least one modified function $f$ and at least one of $f$'s behavior-affected functions, which can be reached from $f$. If they do, they will be put into the retest test suite. This selection algorithm will eliminate more test cases from the retest test pool, so it is more precise. Also, test cases which are not included in the retest test pool are those that do not change the output of the system based on the discussion in the previous section. So the algorithm reserves the safeness also.

**Algorithm 2:** Selection Algorithm by using FCG

**Input:**
- Original Program P
- Modified version P’
- it’s AFDG.
- T: test suite used to test program P
- Each test case has its own FCG.

**Output:**
- T’: the subset of T which includes those test cases need to be retested.

**Step 1:** $T’ = \{\}

For each test case $t_i$ in test suite T, do the following:

Based on FCG, for function $f_i$ in a class, find all other functions within the same class, $f_j$ can reach.

**Step 2:** Identify affected variables in each function.

For each modified function $f_i$, $S_{f_i}$ = variables are affected by the modifications.

For all other functions $f_{o_i}$, $S_{f_{o_i}} = \Phi$.

For each class:

Worklist = \{All modified functions in the class\};

while (Worklist $\neq \Phi$) {

- select and remove $v_i$ from the worklist;
- For all nodes $v_j$ connected with $v_i$ $S = S_{v_j} - N$-list of $v_j$;
- If($S_{v_i} \cap U$-list of $v_j \neq \Phi$)

Then $S = S \cup A$-list of $v_j$

If ($S$ is not $\subset S_{v_i}$) Then $S_{v_i} = S \cup S_{v_j}$;
- add $v_j$ into the worklist.

}

**Step 3:** Search FCG

For each modified function $f_{t_i}$ which executed in $t_i$:

Use the DFS algorithm to search the AFDG.

During the search, when we extend from $v_i$ to $v_j$:

if $v_j$ is “behavior-affected” and $S_{v_j} \neq \Phi$:

1) If edge $e_{ij} = (v_i, v_j)$ is Call-return edge and $e_{ij} = (v_i, v_j)$ in FCG is also Call-return edge and $v_i$ can reach $v_j$ in FCG, $T’ = T’ \cup \{t_i\}$, stop.

2) If the edge $e_{ij} = (v_i, v_j)$ is Sequence edge and $v_i$ can reach $v_j$ in FCG, $T’ = T’ \cup \{t_i\}$, stop.

End

The FCG for the three test cases of the previous example are depicted in Figure 3. Based on our test case selection algorithm, we will only need to retest test case 3.

$N_{v_i} = \text{Maximum Number of data members in one classes.}$

$N_{f_i} = \text{Maximum Number of functions in one class}$
\(N_{\text{class}}\) = Number of classes in the system.
\(N_{m}\) = Number of modified functions.

Step 1 of the algorithm will identify for function \(f_i\), all other functions within the same class it can reaches. This can be done within \(O(N_{\text{class}}\times N_{f1}^2)\). Step 2 will compute the variables that are really affected in each function during the execution of current test case. The running time of Step 2 is decided by the number of items which will added into the worklist. One function \(f\) will be added into the worklist iff \(S_f\) has increased. According to the algorithm, each function will be added into the worklist at most \(O(N_f)\) times. The time we need to process each item popped from the worklist will be \(O(N_{v1}\times N_{f1})\). So the total running time for step1 is \(O(N_{v1}\times N_{f1}^2\times N_{\text{class}})\). The total running time is approximately \(O(N_f^2)\).

Step3 is to check edges in the AFDG, where we will check each edge at most once, so the running time of this step depends on how many edges we need to check in the AFDG. There are at most \(O(N_f^2)\) edges, allowing it to be done within \(O(N_{m}\times N_f^2)\). Thus, the running time of this selection algorithm is \(O(N_{m}\times N_f^2)\).

### 4 Empirical Studies

To demonstrate the strengths of the selection technique and to investigate the effect of different types of modifications on the performance of this technique, we conducted experiments on two systems which were implemented using C++ and which contain a number of natural faults.

#### 4.1 Description

The first system used in this study is an ATM simulator, which is a course project implemented by one of the authors of this paper. This system is described in Table 1, where the version of the system used, version 1, is the system obtained after a function test; the system consists of 43 functions and 1,167 lines of code and the number of test cases used in the function test is 180. During the functional testing, three faults were detected and the program was modified to correct the faults and regression testing was performed on the modified program. To demonstrate the strength of our selection technique, two experiments were conducted in parallel: one is retesting all and the other is a selective retesting, based on the algorithm described in Section 4.2.2.

The results of the regression testing are summarized in Table 2 where Version 2-i denotes the version of the program after the \(i^{th}\) faults corrected, \(i = 1, 2\) and 3, respectively, and Version 2-4 is the program in which all three faults were corrected. The number of modified functions and the number of lines that contain the modified codes are listed in the second and the third columns. The number of test cases selected based on our method are 40, 39, 92 and 112, respectively, which indicates that 78%, 78%, 49% and 38% of the efforts are saved. The last column presents the number of test cases. When executed before and after the modification, the system exhibits different behaviors, i.e. the number of fault-revealing test cases.

The second system we used in this empirical study is a subsystem of an HMI(Human Machine Interface) software that is a fully networked Supervisory Control and Data Acquisition system. This software, consisting of more than 200 subsystems and 3 million lines of code, has been used for several years by numerous manufacturing companies. The subsystem we selected is a communication-oriented program which runs to approximately 16,000 lines of code and 215 functions. The number of test cases used in the function test is 499. This data is presented in Table 3.

After two fault correction activities we obtained two versions of the program, where Version 2 contains 16 modified functions, and 375 lines of code changed. 107 test cases were selected using our method and yielding a 79% saving of effort compared with the retest all approach which used 499 test cases. Version 3 is the program obtained after the second correction activity, which contains two modified functions and 15 lines of code changed. By using our

#### Table 1. The ATM simulator.

<table>
<thead>
<tr>
<th>Versions</th>
<th>#functions</th>
<th>Lines of Codes</th>
<th>Test pool size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version 1</td>
<td>43</td>
<td>1,167</td>
<td>180</td>
</tr>
</tbody>
</table>

#### Table 2. regression testing on ATM simulator.

<table>
<thead>
<tr>
<th>Versions</th>
<th>Ver 2-1</th>
<th>Ver 2-2</th>
<th>Ver 2-3</th>
<th>Ver 2-4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified functions</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Lines of Code changed</td>
<td>3</td>
<td>8</td>
<td>7</td>
<td>18</td>
</tr>
<tr>
<td>Retest test cases</td>
<td>40</td>
<td>39</td>
<td>92</td>
<td>112</td>
</tr>
<tr>
<td>Effort saving</td>
<td>78%</td>
<td>78%</td>
<td>49%</td>
<td>38%</td>
</tr>
<tr>
<td>Faults reveal</td>
<td>33/33</td>
<td>16/16</td>
<td>83/83</td>
<td>105/105</td>
</tr>
</tbody>
</table>

#### Table 3. The subsystem of HMI software.

<table>
<thead>
<tr>
<th>Versions</th>
<th>#functions</th>
<th>#classes</th>
<th>LC</th>
<th>Test pool size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version 1</td>
<td>215</td>
<td>27</td>
<td>16,000</td>
<td>499</td>
</tr>
</tbody>
</table>
Table 4. Regression testing on the HMI subsystem.

<table>
<thead>
<tr>
<th>Versions</th>
<th>Version 2</th>
<th>Version 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified functions</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>Lines of Code changed</td>
<td>375</td>
<td>15</td>
</tr>
<tr>
<td>Retest test cases</td>
<td>107</td>
<td>57</td>
</tr>
<tr>
<td>Effort saving</td>
<td>79%</td>
<td>89%</td>
</tr>
<tr>
<td>Faults reveal</td>
<td>47/47</td>
<td>24/24</td>
</tr>
</tbody>
</table>

method, 57 test cases were selected giving an effort saving of 89%. These results are summarized in Table 4.

From these tables, we observed that: (1) our method is safe. In all the cases, the fault-revealing test cases selected in our method are the same as those included in the retest all method. (2) With a reasonable expense, our method can save a great deal of effort under most circumstances. In 4 out of 6 cases, we have a effort saving of more than 78% each, but many characteristics behind these numbers need to be pulled out. (1) In Version 2-2 and 2-3 of the ATM system. Although the modifications look similar, the results were totally different, one with an effort saving 78% vs. the other one with only 49%. We discovered that the second error was a data management error while the third was an error related to user interface. Because most of the functions which handle user interface are behavior-affected, we can not make use of the advantage of our method with version 2-3. With version 2-2, on the contrary, if the error is a data related error, our method allows us to save a significant effort by the use of function dependence relationship. (2) In Version 2 and 3 of the HMI system, we figure out that the more changes we make, the more affected functions and function dependence relationships, the smaller the amount of effort we save. (3) Many of the code changes in Version 2 of the HMI system involve communications; on the other hand, Version 3 does not involve too many communication-related changes. Because FCG cannot handle distributed systems, so any change in communication needs to be retested. Thus, we obtained less saving in Version 2 than in Version 3.

4.2 Comparison

In this section we compare our method with the existing approaches: Execution slicing, Relevant slicing, firewall method and the method that is based on the control flow graph proposed by Rothermel et al,[16, 17, 18].

Execution slicing:

In Table 5 we present the results obtained from these two methods.

Although Execution slicing is a little faster than our method, precision is a major problem. In version 2-2 of the first system, the difference between execution slicing and our method is 116 to 39. In version 3 of the system2, the result is 89 vs. 57.

Relevant slicing:

Relevant slicing will consider all the predicate statements, so it’s safe and precise. But a program dependence graph must be built for each test case, and then backwards slicing must be done for each PDG. The running time for building the PDG is $O(n^2)$ where $n$ is the number of statements in the system; so this method is often impractical.

Rothermel’s method:

For Rothermel’s method, the basic algorithm is similar to execution slicing, with running time $O(\min(n, n'))$, where $n$ is the number of statements in the original program and $n'$ is the number of statements in the new program. The algorithm with added precision can find the retest suite more precisely, but with a running time of $O(n^2)$, which is very expensive.

In sum, our method is as safe as Execution slicing, Relevant slicing and Rothermel’s method. At the same time, it is more precise than Execution slicing; and more efficient than Relevant slicing or Rothermel’s method.

Firewall method:

Firewall approaches are proposed by Leung and White[20], Kung[12], Hsia[11] and Abdullah[1]. Firewall method first identifies a firewall within which are all modules that may be affected by the modification. The firewall is established according to the inheritance, aggregation and association relations among classes or based on the control-relations among modules. Then the firewall method will choose test cases which exercise at least one of the modules within the firewall. Compared with firewall method, our approach can find more precise function dependence relationship, and perform further analysis for each test case. Therefore our method is more precise than firewall method.
5 Conclusions

We have presented a selective regression technique based on a function dependence relationship. Our approach properly uses a high level of abstraction instead of involving the complex relationship among statements. Therefore, it can achieve a good compromise between efficiency and precision. Moreover, we use the Function Calling Graph to represent the execution history of each test case, which can be processed efficiently and precisely. Our method can also provide useful information to facilitate the designing of new test cases.

References


