RT
256
C.1

Implementation of a Regression Test Selection Technique for C++ Software

By

RANIA M. HAJJAR

Submitted in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science

Thesis Advisor: DR. NASH'AT MANSOUR

Computer Science Program
LEBANESE AMERICAN UNIVERSITY
Beirut, Lebanon
June 2001

LEBANESE AMERICAN UNIVERSITY

GRADUATE STUDIES

We hereby approve the project of

Rania M. Hajjar

candidate for the Master of Science degree* in Computer Science.

Approved by Dr. Nash'at Mansour Signatures Redacted

Associate Professor of Computer Science

Dr. Ramzi Haraty Signatures Redacted

Assistant Professor of Computer Science

Date: July 4, 2001

*We also certify that written approval has been obtained for any proprietary material contained therein.

To My Dear Father and Mother

Table of Contents

CHAPTER 1: INTRODUCTION **1**

CHAPTER 2: REGRESSION TEST SELECTION FOR OBJECT-ORIENTED SOFTWARE .. **6**

2.1 BACKGROUND 6
2.2 REGRESSION TESTING 6
2.3 REGRESSION TESTING OBJECT-ORIENTED SOFTWARE 8

CHAPTER 3: DESCRIPTION OF REGRESSION TEST SELECTION TECHNIQUE **11**

3.1 CONTROL FLOW GRAPH 11
3.2 INTERPROCEDURAL CONTROL FLOW GRAPHS 11
3.3 CODE INSTRUMENTATION 12
3.4 THE OVERALL APPROACH 13
3.5 A REGRESSION TEST SELECTION ALGORITHM 15

CHAPTER 4: DESIGN OF THE IMPLEMENTED TECHNIQUE **19**

4.1 STRUCTURE CHART 19
4.2 STRUCTURE TYPE, CONSTANT AND VARIABLE DEFINITIONS 19
4.3 DETAILED FUNCTIONS DESIGN 21

CHAPTER 5: EXPERIMENTAL RESULTS **27**

CHAPTER 6: CONCLUSION **56**

BIBLIOGRAPHY **58**

APPENDIX: IMPLEMENTATION DETAILS **60**

A. CREATION OF THE CONTROL FLOW GRAPH 60
B. FILES FORMAT 60
B. FILE SPECIFICATIONS AND RULES 61

List of Figures

Figure 3.5.1. Algorithm for test selection for object-oriented software. 18
Figure 4.1. Structure chart of the SelectTest program. 19
Figure 5.1. (a) Actual code for the base version of the Elevator application. 31
Figure 5.1. (b) ICFG for the base version of the Elevator application. 32
Figure 5.1. (c) Input file containing ICFG information for the based version of the Elevator application. 33
Figure 5.1. (d) Actual code for the modified version of the Elevator application. 34
Figure 5.1. (e) ICFG for the modified version of the Elevator application. 35
Figure 5.1. (f) Input file containing ICFG information for the modified version of the Elevator application. 36
Figure 5.1. (g) Test history information table for Elevator application. 37
Figure 5.1. (h) Test(s) selected to be reexecuted on the modified version of the Elevator application. 37
Figure 5.2. (a) Actual code for the base version of the Time application. 39
Figure 5.2. (b) ICFG for the base version of the Time application. 40
Figure 5.2. (c) Input file containing ICFG information for the base version of the Time application. 41
Figure 5.2. (d) Actual code for the modified version of the Time application. 42
Figure 5.2. (e) ICFG for the modified version of the Time application. 43
Figure 5.2. (f) Input file containing ICFG information for the modified version of the Time application. 44
Figure 5.2. (g) Test history information table for Time application. 45
Figure 5.2. (h) Test(s) selected to be reexecuted on the modified version of the Time application. 45
Figure 5.3. (a) Actual code for the base version of the Publication application. 48
Figure 5.3. (b) ICFG for the base version of the Publication application. 49
Figure 5.3. (c) Input file containing ICFG information for the base version of the Publication application. 50
Figure 5.3. (d) Actual code for the modified version of the Publication application. 52
Figure 5.3. (e) ICFG for the modified version of the Publication application. 53
Figure 5.3. (f) Input file containing ICFG information for the modified version of the Publication application. 54
Figure 5.3. (g) Test history information table for Publication application. 55
Figure 5.3. (h) Test(s) selected to be reexecuted on the modified version of the Publication application. 55

Acknowledgements

Above all I would like to express my gratitude to God whom I owe everything I achieve.

I would like to thank Dr. Nash'at Mansour, my advisor, for giving me the necessary guides and advises through my project. Also, I would like to thank Dr. Ramzi Haraty, my second reader, for taking the time to read my project and for his precious comments.

I greatly thank the chairman of the Natural Science Division at LAU, Dr. Ahmad Kabbani, for the support and encouraging he always offer.

Special thank to my family, especially my mother whose love and support kept me going on to complete my higher education; my father who supported and encouraged me in all my study period; my grand-mother for her unceasing prayers and my two sisters Rima and Rola for standing by me throughout all my studies and bearing with me in hard times.

ABSTRACT

by

Rania M. Hajjar

Regression testing is an important but expensive software maintenance activity performed with the aim of providing confidence in modified software. Regression test selection techniques reduce the cost of regression testing by selecting tests for a modified program from a previously existing test suite. Many researchers have addressed the regresion test selection problem for procedural language software, but few have addressed the problem for object-oriented software.

We present an implementation and experimental evaluation of a technique proposed by Rothermel, Harrold, and Dedhia. This technique is a regression test selection technique for object-oriented C++ software. The technique constructs graph representation for software, and uses these graphs to select tests, from the original test suite, that execute code that has been changed for the new version of the software. The technique is strictly code based, and requires no assumptions about the approach used to specify or test the software initially. The technique applies to applications programs that use modified classes. The results indicate that the implemented technique can reduce the number of regression tests that must be run.

Chapter 1: Introduction

Software maintenance activities can account for as much as two-thirds of the overall cost of software production. One necessary activity, regression testing, is performed on modified software to provide confidence that the software behaves correctly and that modifications have not adversely impacted the software's quality. Regression testing plays an integral role in maintaining the quality of subsequent releases of software as that software undergoes maintenance. An important difference between regression testing and development testing is that during regression testing, an established suite of tests may be available for use. By reusing such test suites to retest modified programs testers can reduce the effort required to perform that testing. However, test suites can be large, and the time and effort required to rerun all tests in an existing test suite may be expensive. In such cases, testing efforts must be restricted to a subset of the test suite. The problem of choosing an appropriate subset of an existing test suite is the regression test selection problem; a technique for solving this problem is a regression test selection technique.

Few researchers have addressed the regression test selection problem for object-oriented software (e.g., [2, 4, 11, 12]); even though it is known that the emphasis on code reuse in the object-oriented paradigm both increases the cost of regression testing, and provides greater potential for obtaining savings by using regression test selection techniques. When a class is modified, the modification may impact every applications program that uses that class and every class derived from that class; ideally, every such program and derived class should be retested. The object-oriented paradigm also raises different concerns for regression test selection algorithms. For example, because most classes consist of small interacting methods, regression test selection techniques for object-oriented software must function on interacting routines, that is, at the interprocedural level.

M. Lee, A. J. Offut, and R. T. Alexander, in reference [7], developed a technology to identify potential impacts for object-oriented software before making a change. Software testers can use it to find which areas are impacted by changes during regression testing, enabling them to focus only on those areas and still feel confident about the quality of the software.

Lee J. White, in reference [15], has determined ways to improve both the design and maintenance of OO-systems, including testing and regression testing. A construction of a firewall is done to enclose the set of modules that have been changed and thus must be retested. The firewall is an imaginary boundary that limits the amount of retesting for modified software containing possible regression errors introduced during modification. This procedure also involves the selection or development of test data for regression testing of this change. As long as unit and integration testing are reliable, it is shown that the firewall regression tests are also reliable.

S. Elbaum, D. Gable and G. Rothermel, in reference [1], design an empirical study that allowed them to manipulate and measure various potential sources of variation and prioritization techniques. Test case prioritization techniques assist with test suite reuse by helping testers order their test cases such that those with higher priority, according to some criterion, are executed earlier than those with lower priority.

M. Winter, in reference [14], introduces the class message diagram, a new diagram for object-oriented software combining the behavioral and structural aspects influencing integration and regression testing.

D. Kung, J. Gao, and P. Hsia, in reference [13], proposed the following methods: change analysis to identify affected components; retest strategy generation to produce a cost-effective class test order; test case reuse, modification and generation; regression test plan implementation to retest the modified program.

G. Rothermel, M. Harrold, and J. Dedhia, in reference [13], presented a regression test selection technique that addresses the regression test selection problem for C++ software. The technique constructs control flow representations for classes and programs that use classes; it uses those representations to select tests, from an existing test suite, that execute code that has been changed for a new version of the software.

In reference [12], Rohermel and Harrold presented algorithms for selecting regression tests for C++ software based on walks of program dependence graphs. The algorithm presented in reference [12], like the presented algorithms here, apply to C++ applications programs, classes, and derived classes. Construction of program dependence graphs, however, is more expensive than construction of control flow graphs, thus the algorithm presented here is more efficient than those of reference [12].

In reference [4] and [2], Kung et al. and Hsia et al. present a technique for selecting regression tests for class testing. Their approach is based on the concept of firewalls defined originally by Leung and White for procedural software [8, 9, 13]. Their technique, called ORD technique, constructs an object relation diagram (ORD) that describes static relationship among classes. The ORD technique instruments code to report the classes that are exercised by test cases. The ORD technique and the presented technique are similar in that they both select all tests associated with some set of code components, and the association of tests with code components is determined dynamically through instrumentation. The primary difference between the techniques is the granularity at which they consider components. The ORD technique selects all tests associated with classes within the firewall; it performs no further analysis within classes and methods to attach test to entities at a finer granularity. Not all of those tests necessarily execute changed code, or code that accesses changed data objects. Whereas, the presented technique is more precise than the ORD technique, as it will be shown subsequently.

The presented technique has several advantages. It can be fully automated, it operates interprocedurally, and it performs test selection for C++ applications programs, classes, and derived classes. The technique handles both structural and nonstructural program modifications and processes multiple modifications with a single application of the algorithm. The approach selects tests that may now execute new or modified code, and tests that formerly executed code that has been deleted from the original program. The technique selects tests using information gathered by code analysis but without requiring software specifications, and it is independent of the approach used to generate tests initially for programs and classes.

The next chapter provides an explanation of the regression testing selection for object-oriented software. Chapter 3 describes the basic algorithm for selecting regression tests for modified C++ applications programs. Chapter 4 provides the design of the implemented technique. Chapter 5 describes empirical studies and results in which the technique is applied on three C++ applications software. Finally, chapter 6 presents conclusions.

Chapter 2: Regression Test Selection for Object-Oriented Software

2.1 Background

The following notation is used throughout this report. Let P be a method, class, or program, let P′ be a modified version of P, and S and S′ be the specifications for P and P′, respectively. Let T be a set of tests (a test suite) created to test P.

2.2 Regression Testing

Previous research on regression testing has addressed many topics, including test environments and automation, capture playback mechanisms, test suite management [4], program size reduction, and regression testability. Most recent research on regression testing, however, concerns selective retest techniques [6, 9, 11, 16].

Selective retest techniques reduce the cost of regression testing by reusing existing tests, and identifying portions of the modified program or its specification that should be tested. Selective retest techniques differ from the retest all technique, which runs all tests in the existing test suite. Leung and White [10] show that a selective retest technique is more economical than retest all technique only if the cost of selecting a reduced subset of tests is less than the cost of running the tests that the selective retest techniques omits.

A typical selective retest technique proceeds as follows:

1. Select $T' \subseteq T$, a set of tests to execute on P'.
2. Test P′ with T′, establishing P′'s correctness with respect to T′.

3. If necessary, create T″, a set of new functional or structural tests for P′.
4. Test P′ with T″, establishing P′'s correctness with respect to T″.
5. Create T‴, a new test suite and test history for P′, from T, T′, and T″.

In performing these steps, a selective retest technique addresses several problems. Step 1 involves the regression test selection problem: the problem of selecting a subset T′ of T with which to test P′. This problem includes the subproblem of identifying tests in T that are obsolete for P′. Test t is obsolete for P′ if t specifies input to P′ that, according to S′, is invalid for P′, or t specifies an invalid input-output relation for P′. Step 3 addresses the coverage identification problem: the problem of identifying portions of P′ or S′ that require additional testing. Step 2 and 4 address the test suite execution problem: the problem of efficiently executing tests and checking test results for correctness. Step 5 addresses the test suite maintenance problem: the problem of updating and storing test information. Although each of these problems is significant, this project restricts its contents to the regression test selection problem. Furthermore, it restricts its contents to code based regression test selection techniques, which rely on analysis of P and P′ to select tests.

There are two distinguishable phases of regression testing: a preliminary phase and a critical phase. The preliminary phase begins after the release of some version of the software; during this phase, programmers enhance and correct the software. When corrections are complete, the critical phase of regression testing begins; during this phase regression testing is the dominating activity, and its time is limited by the deadline for product release. It is in the critical phase that cost minimization is most important for regression testing.

Regression testing techniques can exploit these phases. For example, a technique that requires test history and program analysis during the critical phase can achieve a lower critical phase cost by gathering that information during the preliminary phase.

There are various ways in which this two-phase process may fit into the overall software maintenance process. A big bang process performs all modifications, and when these are complete proceeds with regression testing. An incremental process performs regression testing at intervals throughout the maintenance life cycle, with each testing session aimed at the product in its current state of evolution. Preliminary phases are typically shorter for the incremental model than for the big bang model; however, for both models, both phases exist and can be exploited.

2.3 Regression Testing Object-Oriented Software

Object-oriented languages raises interesting concerns for regression testing. To test object-oriented software properly, we require a technique for testing classes. Class testing approaches typically invoke sequences of methods in varying orders, and after each sequence, verify that the resulting state of the object manipulated by the methods is correct.

The object-oriented paradigm provides new applications for selective retest algorithms. When we modify a class, we must retest the class, and classes derived from that class. Moreover, although encapsulation should reduce the likelihood that object-oriented code modules will interact inappropriately, it is still the case that tests run on applications programs may reveal faults in methods that were not revealed by tests of the individual methods. Thus, if we want to be sure that we have rerun all existing tests that may expose errors in a modified class (i.e., select a safe test) we must consider all applications programs that use the modified class. Whether retesting applications programs, classes, or derived classes, we can benefit by applying selective retest algorithms to existing test suites.

To perform class testing, we require a driver that invokes a sequence of methods. A typical class test driver first performs "setup" chores, calling constructor routines and/or other methods. Next, the driver invokes the sequence of methods under test. Finally, the driver invokes an "oracle" method that verifies that objects have attained proper states. Code-based selective retest methods must be able to distinguish between drivers, setup, routines, and methods actually under test, and select only the tests that are relevant to changes in methods under test.

When we modify a class, we must retest the class and classes derived from that class. Unfortunately, it has been shown that a function that has been adequately tested in isolation may not be adequately tested in combination with other functions. Thus, it is advisable to also test applications programs that use the modified class. In practice, of course, it may be impractical or impossible to retest all such applications programs; nevertheless, we need to be aware of the consequences of this impracticality, and we may wish to reduce the associated risks by regression testing those applications programs that can economically be tested.

Object-oriented programs employ polymorphism and dynamic binding to a degree beyond that of procedural programs. In an object-oriented program a method invocation can be bound at run-time to a number of methods. For a given calls, we cannot always determine statically the method to which it will be bound. Selective retest methods that rely on static analysis must provide mechanisms for coping with this uncertainty.

Chapter 3: Description of Regression Test Selection Technique

3.1 Control Flow Graph

A control flow graph (CFG) for method P contains a node for each simple or conditional statement in P; edges between nodes represent the flow of control between statements. In a graph, the statement node that are drawn as ellipses, represent simple statements. Predicate nodes, drawn as rectangle, represent conditional statements. Branches, drawn as labeled edges leaving predicate nodes, represent control paths taken when the predicate evaluates to the value of the edge label. Statement and predicate nodes are labeled to indicate the statements in P to which they correspond. To label nodes, statement numbers can be used in the control flow graph; however, the actual code of the associated statements could also serve as labels. For simplicity, switch statements are represented in CFGs as nested if-else statements; under this assumption, every CFG node has either one unlabeled out-edge or two out-edges labeled "T" and "F". A unique entry node, labeled with a statement number, and a unique exit node, labeled "X", represents entry to and exit from P, respectively. Declarations and initialization statements, when represent, are also represented as nodes. The time and space required to construct and store a CFG for method P is linear in the number of simple and conditional statements in P.

3.2 Interprocedural Control Flow Graphs

A CFG encodes control flow information for a single method. To encode control flow for a group of interacting methods, that have a single entry point, such as a group of methods that constitute an entire program, we use an interprocedural control flow graph (ICFG). An ICFG for program P contains a control flow graph for each method in P, with each call site in P represented as a pair of nodes called call and return nodes. Each call node is connected to the entry node of the called method by a call edge, and each exit node is connected to the return node of the calling method by a return edge.

3.3 Code Instrumentation

Let P be a program with ICFG G. we can instrument P such that when the instrumented version of P is executed with test t, it records a branch trace that consists of the branches taken during that execution. We can use this branch trace information to determine which edges in G were traversed when t was executed: an edge (n_1, n_2) in G is traversed by test t if, when P is executed with t, the statements associated with n_1 and n_2 are executed sequentially at least once during the execution. We call the information thus determined an edge trace for t on P. An edge trace for t on P has size linear in the number of edges in G, and can be represented by a bit vector.

Given test suite for P, we construct a test history for P with respect to T by gathering edge trace information for each test in T and representing it such that for each edge (n_1,n_2) in G, the test history records the tests that traverse (n_1,n_2). This representation requires $O(e|T|)$ bits, where e is the number of edges in G and $|T|$ is the number of tests in T.

3.4 The Overall Approach

Two assumptions are made. First assumption is that T contains no obsolete tests, either because it contained none initially, as in purely corrective maintenance, or because it has been removed. This assumption is reasonable because it is required for regression testing of any form: if test obsolescence cannot be determined then test correctness cannot be judged. Second assumption is that each test in T terminated when run on P, and produced its specified output. This assumption too is reasonable, because if any tests violate it (as in practice they may, because software is often released with known bugs) these tests are known, and can be removed from T prior to performing regression test selection and then reincluded in T′ following the selection phase.

The goal of this approach is to identify tests in T that execute changed code with respect to P and P′. In other words, to identify tests in T that (1) execute code that is new or modified for P′ or (2) executed code in P that is no longer present in P′. To formalize the notion of these tests, a definition is given to an execution trace for test t on P to consist of the sequence of statements in P that are executed when P is executed with t. Two execution traces are equivalent if they have the same lengths and if, when their elements are compared from first to last, the text representing the pairs of corresponding elements is lexicographically equivalent. Two text strings are lexicographically equivalent if their text, ignoring extra white space characters when not contained in character constants, is identical. Test t is modification traversing for P and P′ if its execution traces from P and P′ are nonequivalent.

In general, it may be extremely expensive to compare execution traces, and an algorithm that attempted it would be required to run all tests in T on P′ (which the presented approach is trying to avoid) in order to obtain their traces and select a subset of T. We can take advantage, however, from the mapping between execution traces and paths in ICFGs obtained by replacing each statement in the execution trace by its corresponding ICFG node (or equivalently, by the text of the statement associated with that node) to ensure selection of tests that are modification traversing. The proposed algorithm synchronously traverses ICFG paths that begin with the entry node of G and G′, looking for pairs of nodes N and N′ whose associated statements are not lexicographically equivalent. When such a pair is found, a test history information is used, obtained through the TestsOnEdge function to select all tests known to have reached N. This traversal essentially considers all tests at once; it is not necessary to perform one traversal per test. By marking nodes "visited" as the traversing is taking place, the approach avoids visiting nodes multiple times, and ensures that the algorithm terminates in time proportional to the size of the graph, rather than to the size of the execution traces.

A focus on changes in executable statements is done so far. A change in a nonexecutable variable or type declaration may cause a test to reveal a fault, even though that test executes no changed executable statements. For example, in a C++ program, changing the type of a variable from "int" to "double" can cause the program to fail even if no executable statements are altered. To handle this situation, one approach is to create, in ICFGs, declaration nodes that correspond to variable and type declarations, and associate each test that executes a method, class, or program with all declaration nodes associated with that method, class or program. Following this approach, when nonexecutable initialization statements change, SelectTests selects all tests attached to the associated nodes.

3.5 A Regression Test Selection Algorithm

The proposed regression test selection algorithm, SelecTests, is presented in figure 3.5.1 takes a program P, its modified version P′, and the test suite T for P, and returns T′, a set that contains tests that are modification traversing for P and P′. SelectTests first initializes T′ to ϕ, and initializes E, which will hold the set of edges in the ICFG for P on which tests must be selected, to ϕ. Next, the algorithm constructs ICFGs G (with entry node e) and G′ (with entry node e′) for P and P′, respectively. The algorithm then calls Compare with e and e′. Compare ultimately places edges through which tests become modification traversing for P and P′ into E. SelecTests then uses TestOnEdges to retrieve these tests (line 6-7).

Compare is a recursive procedure called with pairs of nodes N and N′, from G and G′, respectively, that are reached simultaneously during the algorithm's comparisons of traversal trace prefixes. Given two such nodes N and N′, Compare first determines whether the two nodes have equivalent outgoing edges. If not, then tests that reach N′ may become modification traversing. Thus, Compare selects all edges out of N: the tests on these edges include all tests of interest. Note that if N and N′ represent typical true-valued of false-valued predicate statements, then each node necessarily has a pair of outgoing edges labeled "true" and "false", and Compare will not need to select any edges.

If N and N′ have equivalent outgoing edges, Compare considers successor nodes of N and N′ to determine whether N and N′ have successor nodes whose labels differ along pairs of identically labeled edges. If N and N′ have any such successors, tests that traverse the edges to the successor are modification traversing due to changes in the code associated with those successors. In this case, Compare selects the edge in G that connects N to that successor. If N and N′ have successors whose labels are the same along a pair of identically labeled edges, Compare continues along the edges in G and G′ by invoking itself on those successors.

Lines (10-29) of Figure 3.5.1 describe Compare's actions more precisely. When Compare is called with ICFG nodes N and N′, Compare first marks node N "N′-visited"(line 11). After Compare, has been called once with N and N′ it does not need to consider them again – this marking step lets Compare avoid revisiting pairs of nodes. Lists to hold "visited" information are associated with ICFG nodes in G, and created and initialized during ICFG construction. Next, Compare uses OutEdgesEquivalent(N,N′) to check for equivalence of outgoing edges. This function returns true only if there is one-to-one correspondence between the labels on edge leaving N and the labels on edges leaving N′; if there is no such correspondence the function returns false and lines (13-15) select the necessary edges. If N and N′ have equivalent outgoing edges, Compare, in the for loop of lines (17-27), considers each control flow successor of N. for each successor C, Compare locates the label L on the edge from N to C, then seeks the node C′ in G′ such that (N′,C′) has label L; if (N,C) is unlabeled, ε is used for the edge label. Next, Compare considers C and C′. if C is marked "C′-visited", Compare has already been called with C and C′, so

Compare does not take any action with C and C′. if C is not marked "C′-visited", Compare calls NodesEquivalent with C and C′. The NodesEquivalent function takes a pair of nodes N and N′, and determines whether the statement S and S′ associated with N and N′ are lexicographically equivalent. If NodesEquivalent(C,C′) is false, then tests that traverse edge (N,C) are modification traversing for P and P′, and tests that executed C must be selected; Compare adds edge (N,C) to E, the list of edges on which tests will be selected. If NodesEquivalent(C,C′) is true, Compare invokes itself on C and C′ to continue the graph traversals beyond these nodes.

SelectTests is an interprocedural algorithm: it functions on entire programs rather than single methods. By beginning with the entry nodes of "main" routines, and processing called methods only when it reaches calls to those methods, SelecTests avoids analyzing methods called only after code changes.

```
Algorithm SelectTests(P,P′,T):T′
Input       P,P′: base and modified versions of a program
            T: a test set used to test P
Output      T′: the subset of T selected for use in regression testing P′
Global      E: a subset of the edges in the ICFG for P

1.   begin
2.        $T' = \phi$
3.        $E = \phi$
4.        construct G and G′, ICFGs for P and P′, with entry nodes e and e′
5.        Compare(e,e′)
6.        for each edge $(n_1,n_2) \in E$ do
7.             $T' = T' \cup$ TestsOnEdge($(n_1,n_2)$)
8.        return T′
9.   end


procedure Compare(N, N′)
input     N and N′: nodes in G and G′

10. begin
11.  mark N "N′-visited"
12.  if not OutEdgesEquivalent(N, N′)
13.       for each successor C of N in G do
14.            $E = E \cup (N, C)$
15.       endfor
16.  else
17.       for each successor C of N in G do
18.            L = the label on edge (N,C) or $\in$ if (N C) is unlabeled
19.            C′ = the node in G′ such that (N′, C′) has label L
20.            if C is not marked "C′-visited"
21.                 if not NodesEquivalent(C, C′)
22.                      $E = E \cup (N, C)$
23.                 else
24.                      Compare(C, C′)
25.                 endif
26.            endif
27.       endfor
28.  endif
29 end
```

Figure 3.5.1. Algorithm for test selection for object-oriented software.

Chapter 4: Design of The Implemented Technique

4.1 Structure Chart

The structure chart of the SelectTest program is in Figure 4.1.

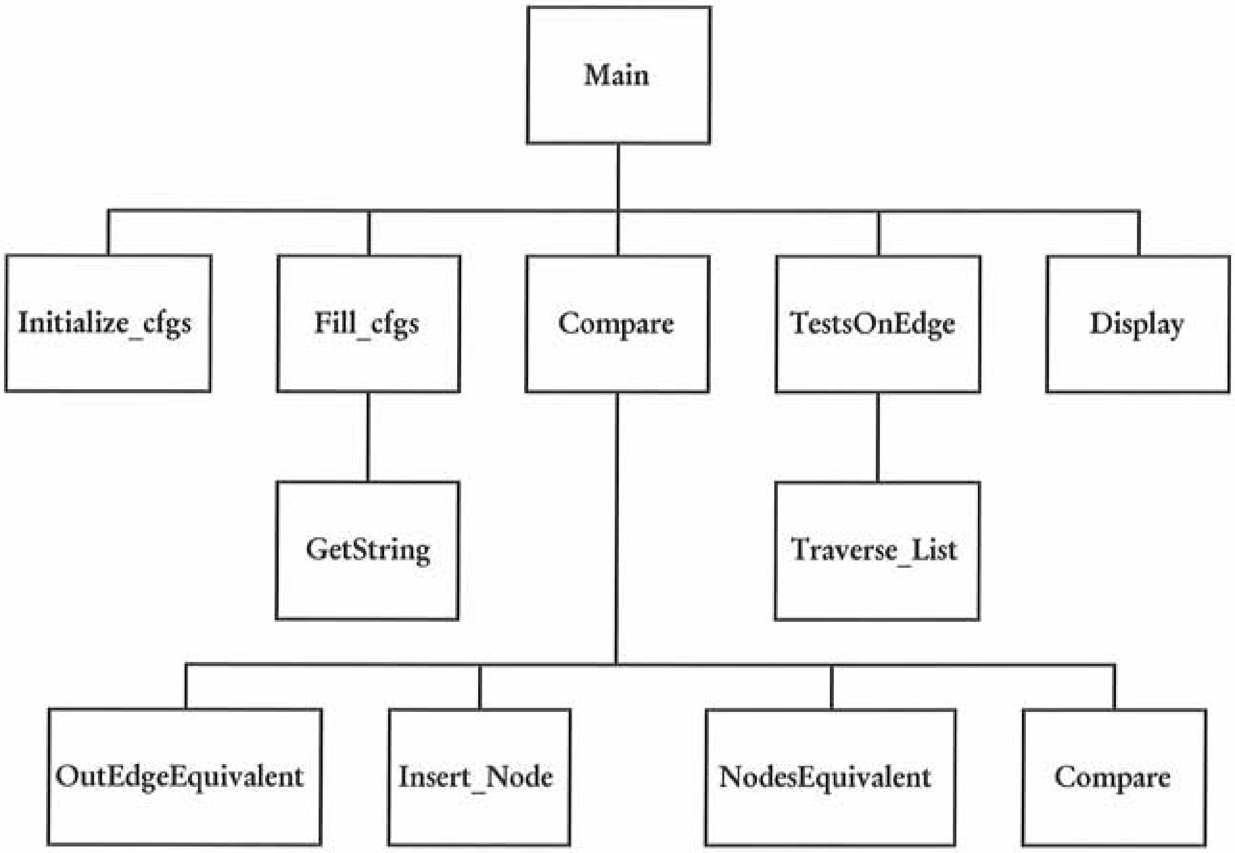

Figure 4.1. Structure chart of the SelectTest program.

4.2 Structure Type, Constant and Variable Definitions

ORG_NB_ND: constant representing the total number of nodes in the interprocedural control flow graph of the original version of the program.

MOD_NB_ND: constant representing the total number of nodes in the interprocedural control flow graph of the modified version of the program.
LEN: constant representing the maximum length of a given type that can be read into an array.
Edge: structure type containing three members:

1. **Node1**: integer type representing a node number leaving an edge
2. **Node2**: integer type representing a node number arriving to an edge.
3. **Next**: pointer to edge type.

Node: structure type containing five members:

1. **Outedgelabel**: pointer to character representing the label on the out edge. It is formed of three characters each of which has value either T (for True) or F (for False). These three characters represent label true, false and normal edge respectively. If one of these characters is set to true this means that the corresponding label exists.
2. **Succ1nb**: integer type representing the first successor node number (its value is zero if there is no successor).
3. **Succ2nb**: integer type representing the second successor node number (its value is zero if there is no successor).
4. **Stmt1:** pointer to character representing the actual code associated with the first successor statement.
5. **Stmt2:** pointer to character representing the actual code associated with the second successor statement.

List_of_edges: pointer to structure type edge containing the list of edges on which tests will be selected.
Orgcfg [ORG_NB_ND**]:** variable array of type edge and length ORG_NB_ND containing the ICFG details of the original version of the program.

Modcfg [MOD_NB_ND]: variable array of type edge and length MOD_NB_ND containing the ICFG details of the modified version of the program.

Orgprg: pointer to file type representing the name of the file containing ICFG information about the original version of the program.

Modprg: pointer to file type representing the name of the file containing ICFG information about the modified version of the program.

Testhis: pointer to file type representing the name of the file containing the test history information of the program.

Visited_array [ORG_NB_ND]: variable array of type integer and length ORG_NB_ND used to record the node traversed.

Testnb: variable of type integer containing the number of tests executed on a given program and is deduced from testhis file.

4.3 Detailed Functions Design

A) Main Function

Input: orgcfg, modcfg, visited_array, list_of_edges, testhis, testnb, regtest

Output: tests that are modification traversing and will be used for re-execution

Description: Main reads a program and its modified version in addition to its corresponding test suite and returns a set containing test that are modification traversing for the program and its modified version.

Pseudo-code:

```
initialize_cfg(orgcfg,modcfg).
initialize_int(visited_array).
fill_cfgs(orgcfg,modcfg).
create first edge in list_of_edges and initialize it to zero
call Compare(0,0,list_of_edges).
```

```
open testhis file.
initialize(regtest).
testnb= TestsOnEdge(regtest).
display(list_of_edges,regtest,testnb).
close teshis file.
```

B) Initialize_cfg Function

Input: orgcfg, modcfg.

Output: orgcfg, mmodcfg.

Description: initialize the two arrays of structure type edge for original and modified program.

Pseudo-code:

```
For each entry in orgcfg and modcfg arrays do
    Set member succ1nb & succ2nb to zero.
    Set member outedgelabel to "FFF".
    Set member stmt1 & stmt2 to space.
```

C) Fill_cfgs Function

Input: orgcfg, modcfg.

Output: orgcfg, modcfg.

Description: read the two files that contain the original and modified version of the program in order to fill the corresponding arrays elements with their each values according to the specified format of the file.

Pseudo-code:

```
Open file orgprg (then modprg).
While not end of file do
    Read line from orgprg (modprg).
```

```
Set nodenb to first field in line.
Set outedgelabel member of orgcfg (modcfg) for entry
nodenb to second field in line.
Set succ1nb member of orgcfg (modcfg) for entry nodenb
to third field in line.
Set succ2nb member of orgcfg (modcfg) for entry nodenb
to fourth field in line.
Set stmt1 member of orgcfg (modcfg) for entry nodenb to
fifth field in line.
Set stmt2 member of orgcfg (modcfg) for entry nodenb to
sixth field in line.
```

D) Compare Function

Input: orgnode and modnode, which are nodes from original and modified version of the program, respectively, reached during the algorithm comparison of traversal trace prefixes.

Description: link edges through which tests become modification traversing for original and modified program to list_of_edges.

Pseudo-code:

```
Set visited_array for entry orgnode to modnode.
If (not outEdgeEquivalent(outedgelabel member of orgcfg for
entry orgnode, outedgelabel member of modcfg for entry
modnode))
{
    if (succ1nb member of orgcfg for entry orgnode not =
    zero)
        insert(orgnode, succ1nb member of orgcfg for entry
        orgnode)
    if (succ2nb member of orgcfg for entry orgnode not =
    zero)
        insert(orgnode, succ2nb member of orgcfg for entry
        orgnode)
}
```

```
else
{
    if ((succ1nb member of orgcfg for entry orgnode not =
    zero) and (visited_array for entry succ1nb member of
    orgcfg for entry orgnode not = succ1nb member of
    modcfg for entry modnode))
    {
      if (not NodesEquivalent(stmt1 member of orgcfg for
      entry orgnode, stmt1 member of modcfg for entry
      modnode))
            insert(orgnode, succ1nb member of orgcfg for
            entry orgnode)
      else
            Compare(succ1nb member of orgcfg for entry
            orgnode, succ1nb member of modcfg for entry
            modnode,list_of_edges)
      if (not NodesEquivalent(stmt2 member of orgcfg for
      entry orgnode, stmt2 member of modcfg for entry
      modnode))
            insert(orgnode, succ2nb member of orgcfg for
            entry orgnode)
      else
            Compare(succ2nb member of orgcfg for entry
            orgnode, succ2nb member of modcfg for entry
            modnode,list_of_edges)
    }
}
```

E) OutEdgeEquivalent Function

Input: outEdgeLabelO outEdgeLabelM.

Output: return 1 as True or 0 as False.

Description: this function returns true only if there is one-to-one correspondence between the labels on edges leaving a node from original ICFG program and the label on edge leaving a node from modified ICFG program, if there is no such corrsponding the function returns false.

F) **NodesEquivalent Function**

Input: Ostatement, mstatement.

Output: return 1 as True or 0 as False.

Description: this function takes a pair of nodes from original and modified program, and determines whether the statements associated with each node are lexicographically equivalent. If it returns false then tests that traverse the node in the original program and its successor are modification traversing for the original program and its modified version.

G) **Traverse_list Function**

Input: list_of_edges, node1, node2.

Output: return 1 as True or 0 as False.

Description: this function traverses the list of edges, taken from the ICFG for the original program, through which tests become modification traversing for the modified version of the program.

Pseudo-code:

```
While not end of list_of_edges
    If ((node1 member of list_of edges = node1) and (node2
    member of list_of edges = node2))
        Return 1
    Else
        Move to next edge in list_of_edges
Return 0
```

H) **TestsOnEdge Function**

Input: regtest

Output: testnb

Description: read testhis file and select tests that has edges that are modification traversing for the modified version of the program.

Pseudo-code:

```
Set testnb to zero
While not end of testhis file do
{
    Initialize(edgetrace)
    Read line from testhis & fill it into edgetrace array
    While not end of edgetrace
    {
        Set node1 to first field in edgetrace
        Set node2 to second field in edge trace
        If (traverse_list(list_of_edges, node1, node2))
        {
            Set regtest of entry testnb to 1
            Break
        }
    }
    Increment testnb by one
}
return testnb.
```

Chapter 5: Experimental Results

To experiment the proposed approach, three programs were chosen: Elevator, time, and publication applications. The first application uses the elevator class, the second uses the time class and the last application contains multiple inheritance where class book and class tape are derived from the two base classes publication and sales. Each application has a test suite containing between two to four tests each. Three input files are needed for each application: test history information file, ICFG information file for original and modified version.

Figure 5.1 (a) shows the actual code for the base version of the Elevator application.
The elevator application is formed of one class which is the elevator class with one constructor and one destructor, six member functions and four data items. Figure 5.1 (b) shows the ICFG for the base version of the Elevator application. Statement nodes, shown as ellipses, represent simple statements. Predicate nodes, shown as rectangles, represent conditional statements. Branches, shown as labeled edges leaving predicate nodes, represent control paths taken when the predicate evaluates to the value of the edge label. Statement and predicate nodes are labeled to indicate the statements in the elevator application to which they correspond. Statement numbers are used as node labels.
Figure 5.1 (c) shows the input file containing ICFG information for the base version of the Elevator application. This file is described in details in the appendix. Figure 5.1 (d) shows the same information as (a) but for the modified version of the Elevator application. A member function, which_floor, is added,

statement 40 is altered and statement 44.1 is added. Note that the modified part of each application is highlighted in the figure where the actual code of the modified version of the application is present. Figure (e) shows the same information as (b) but for the modified version of the Elevator application. Because which_floor is not invoked by the modified version of the elevator application, its CFG is not needed or included in the ICFG of figure 5.1 (e). Figure 5.1 (f) shows the same information (c) but for the modified version of the Elevator application. Line 20 and 25 of figure 5.1 (f) differs from line 20 and 25 in figure 5.1 (c) concerning the statements equivalence. Figure 5.1 (g) shows the test history information table for Elevator application. Four tests are executed on the elevator application. We assume that a test driver sets the values of top_floor, bottom_floor, and current_floor as indicated, and then invokes the method go with the indicated value of floor as parameter. For each test, the edge trace information is recorded representing all edges traversed during the execution of the elevator application with the corresponding test. Figure 5.1 (h) shows the selected test(s) to be reexecuted on the modified version of the Elevator application. The table is formed with two columns: the first column represents the result when the above mentioned modification are applied, an edge (36,40) is selected by the program, and two tests t3 and t4 are selected according to this edge; the second column represents the result when only a statement is added (44.1) without altering the other statement (40), an edge (40,45) also is selected and only one test t4 containing that edge is selected accordingly.

To see how SelectTests works, the Elevator application is an example that will be explained in details. The Elevator class (presented in Figure 5.1 (a)) is modified, creating Elevator '; the new and modified code is shown in Figure 5.1 (d)). In Elevator ', the go method has changed: line 40 has been (erroneously) altered, and a new line 44.1, has been added. Also a new method, which_floor, has been added. The test set T for Elevator application contains test t_1, t_2, t_3, and t_4, which are described in Figure 5.1 (g). We wish to select tests from T for reexecution when Elevator application is built with the modified version of the Elevator class, yielding the modified version of Elevator application.

In this case, we call SelectTests with Elevator application, its modified version, and T. SelectTests first constructs the ICFGs G and G' for the two programs. Figure 5.1 (e) depicts G'; G' differs from G only with respect to the CFG for go. Because which_floor is not invoked by modified version of Elevator application, its CFG is not needed or included in the ICFG for the modified version of Elevator application.

Next, SelectTests invokes Compare with entry nodes 74 and 74'. Compare begins to visit pairs of nodes: (74,74'), (75,75'), (77c,77c'), (9,9'), (10,10'), (11,11'), (12,12'), (13,13'), (X,X'), (77r,77r'), (78c,78c'), (30,30'), (31c, 31c'), (61,61'), (62,62'), (63,63'), (64,64'), (X,X'), (31r,31r'), (32,32'), and then (36,36'). On visiting (36,36'), Compare first marks "36'-visited", and then considers the successor of 36, 40. The successor in G' with an equivalently labeled edge ("F") is 40'. 40 is not marked "40'-visited", so Compare proceeds to step 21. 40 and 40' are not lexicographically equivalent; thus, Compare insert edge (36,40) into E. The algorithm does not continue further from 40 and 40'. Instead, it traverses other portions of the ICFGs. No further changes are discovered; on return to the algorithm's main routine, tests t_3 and t_4 are the only tests selected.

If the change at 40 had not been present, the algorithm would have continued its traversal from 40 and 40', found their successors 45 and 44.1 lexicographically different, and selected edge (40,45) (requiring only test t_4).

```
#include <iostream.h>
#include <stdlib.h>
#define UP 1
#define DOWN 2
typedef int Direction;

class Elevator   {
public:
Elevator (int l_top_floor)   {
current_floor = 1;
current_direction = UP;
top_floor = l_top_floor;
bottom_floor = 1;
}

virtual ~Elevator () { }

void up ()       {
current_direction = UP;
}

void down()   {
current_direction = DOWN;
}

Direction direction (){
return current_direction;
}

virtual void go (int floor)   {
int valid = valid_floor(floor);
if (!valid)        {
cout << "Invalid floor request\n";
return;
}
if (floor > current_floor)           {
up();
cout << "Elevator is going up";
}
else { if (floor < current_floor)   {
down();
cout << "Elevator is going down";
}
else
return;
if (current_direction == UP)     {
while (( current_floor != floor) &&
(current_floor <= top_floor))
add(current_floor, 1);
}
else   {
while (( current_floor != floor) &&
(current_floor <= bottom_floor))
add(current_floor, -1);
}
};

private:
void add(int &a, const int &b)
     {
a = a+b;
}

int valid_floor(int floor)              {
if ((floor > top_floor) || (floor <
bottom_floor))
return 0;
return 1;
};

protected:
int current_floor;
Direction current_direction;
int top_floor;
int bottom_floor;
};

void main ()   {
Elevator *e_ptr;

e_ptr = new Elevator(10);
e_ptr->go(2);
}
```

Figure 5.1. (a) Actual code for the base version of the Elevator application.

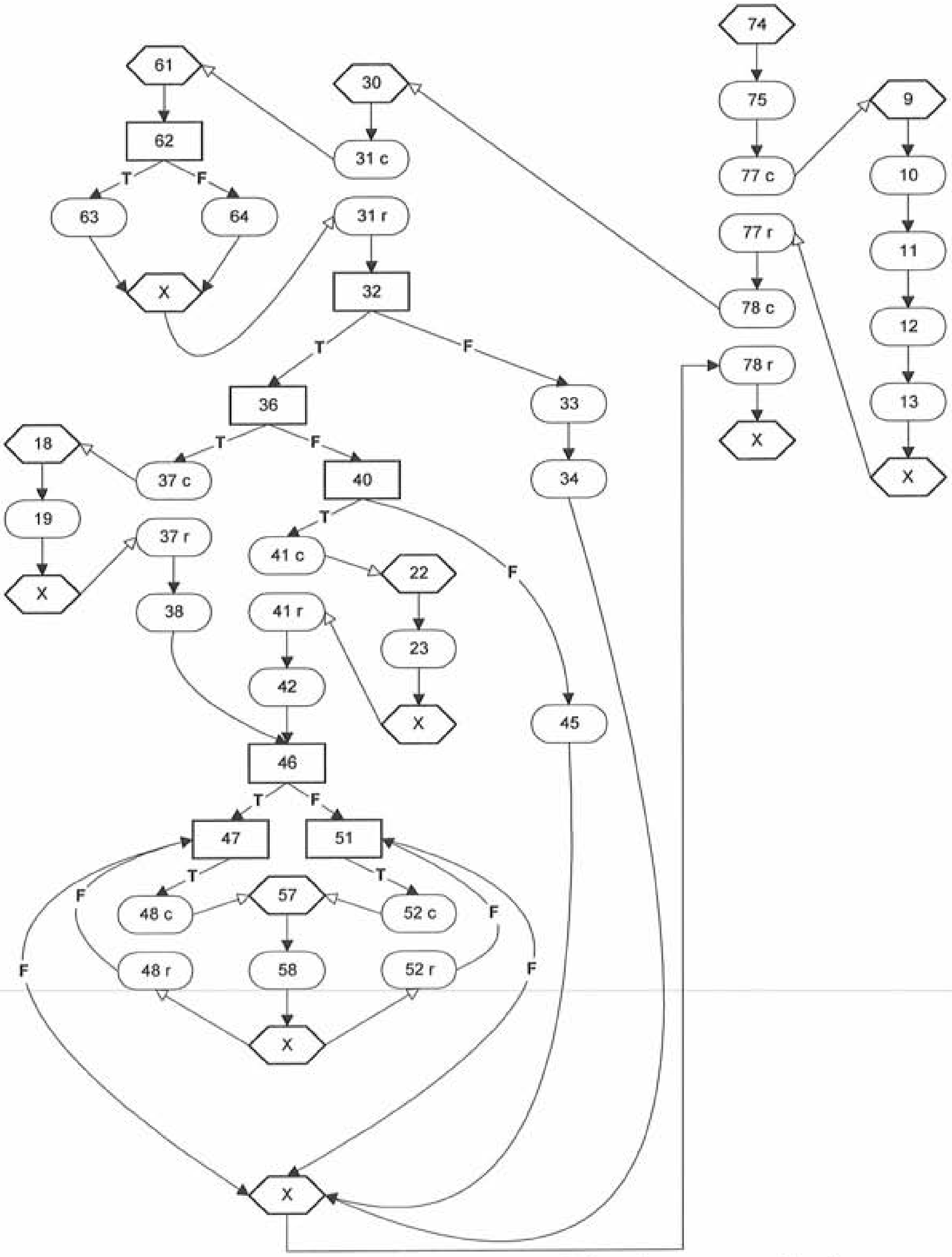

Figure 5.1. (b) ICFG for the base version of the Elevator application.

```
0#FFT#74#0#void main () {# #
74#FFT#75#0#Elevator *e_ptr;# #
75#FFT#77#0#e_ptr = new Elevator(10);# #
77#FFT#9#78#Elevator (int l_top_floor) {#e_ptr->go(2);#
9#FFT#10#0#current_floor = 1;# #
10#FFT#11#0#current_direction = UP;# #
11#FFT#12#0#top_floor = l_top_floor;# #
12#FFT#13#0#bottom_floor = 1;# #
13#FFT#0#0# # #
78#FFT#30#0#virtual void go (int floor) {# #
30#FFT#31#0#int valid = valid_floor(floor);# #
31#FFT#61#32#int valid_floor(floor) {#if (!valid) {#
61#FFT#62#0#if ((floor > top_floor) || (floor < bottom_floor))# #
62#FFT#63#64#return 0;#return 1;#
63#FFT#0#0# # #
64#FFT#0#0# # #
32#TTF#33#36#cout << "Invalid floor request\n";#if (floor > current_floor) {}#
33#FFT#34#0#return;# #
34#FFT#0#0# # #
36#TTF#37#40#up();#else if (floor < current_floor) {#
37#FFT#18#38#void up() {#cout << "Elevator is going up";#
18#FFT#19#0#current_direction = UP;# #
19#FFT#0#0# # #
38#FFT#46#0#if (current_direction == UP) {# #
40#TTF#41#45#down();#return;#
41#FFT#22#42#void down(){#cout << "Elevator is going down";#
22#FFT#23#0#current_direction = DOWN;# #
23#FFT#0#0# # #
42#FFT#46#0#if (current_direction == UP) {# #
46#TTF#47#51#while ((current_floor != floor) && (current_floor <= top_floor))#while
((current_floor != floor) && (current_floor <= bottom_floor))#
47#TTF#48#0#add(current_floor, 1);# #
48#FFT#57#0#add(int &a, const int &b) {# #
57#FFT#58#0#a = a+b;# #
58#FFT#0#0# # #
51#TTF#52#0#add(current_floor, -1)# #
52#FFT#0#0# # #
45#FFT#0#0# # #
%
```

Figure 5.1. (c) Input file containing ICFG information for the based version of the Elevator application.

```
1'  #include <iostream.h>
2'  #include <stdlib.h>
3'  #define UP 1
4'  #define DOWN 2
5'  typedef int Direction;
6'
7'  class Elevator   {
8'  public:
9'  Elevator (int l_top_floor)   {
10' current_floor = 1;
11' current_direction = UP;
12' top_floor = l_top_floor;
13' bottom_floor = 1;
14' }
15'
16' virtual ~Elevator () { }
17'
18' void up ()        {
19' current_direction = UP;
20' }
21'
22' void down()    {
23' current_direction = DOWN;
24' }
25'
26' Direction direction (){
27' return current_direction;
28' }
28.1    int which_flooor( )              {
28.2        return current_floor;}
29'
30'  virtual void go (int floor)   {
31' int valid = valid_floor(floor);
32' if (!valid)         {
33' cout << "Invalid floor request\n";
34' return;
35' }
36' if (floor > current_floor)           {
37' up();
38' cout << "Elevator is going up";
39' }
40'     else { if (floor <=
current_floor)      {
41'                 down();
42'  cout << "Elevator is going
down";
43'                  }
44'                  else   {
44.1     cout<< "Elevator is on the
         same floor\n";
45'  return;  }
46  if (current_direction == UP)      {
47  while (( current_floor != floor) &&
    (current_floor <= top_floor))
48  add(current_floor, 1);
49  }
50  else   {
51  while (( current_floor != floor) &&
    (current_floor <= bottom_floor))
52  add(current_floor, -1);
53  }
54  };
55
56  private:
57  void add(int &a, const int &b)
        {
58  a = a+b;
59   }
60
61  int valid_floor(int floor)            {
62  if ((floor > top_floor) || (floor <
    bottom_floor))
63  return 0;
64  return 1;
65  };
66
67  protected:
68  int current_floor;
69  Direction current_direction;
70  int top_floor;
71  int bottom_floor;
72  };
73
74  void main ()    {
75  Elevator *e_ptr;
76
77  e_ptr = new Elevator(10);
78  e_ptr->go(2);
79  }
```

Figure 5.1. (d) Actual code for the modified version of the Elevator application.

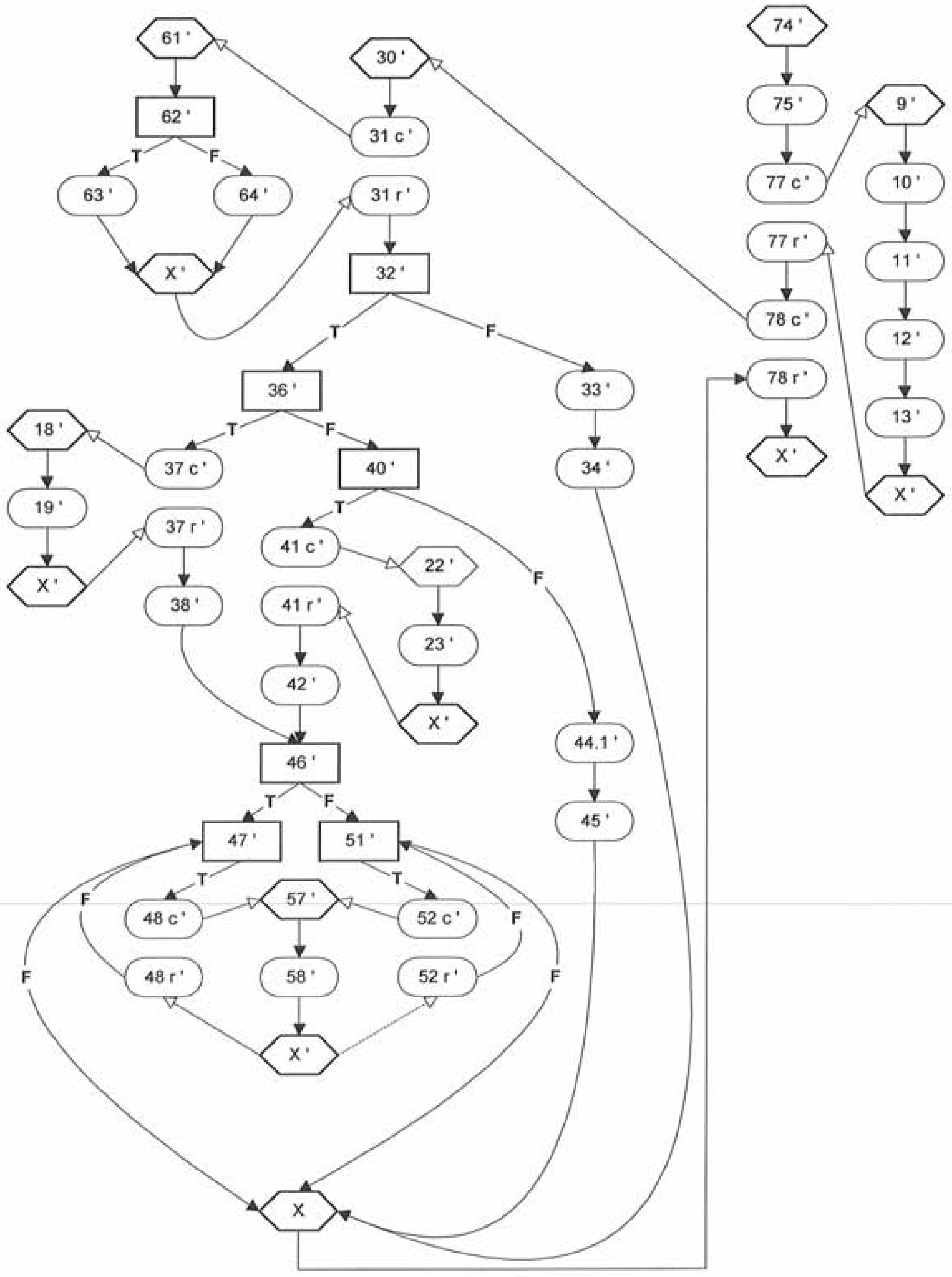

Figure 5.1. (e) ICFG for the modified version of the Elevator application.

```
0#FFT#74#0#void main () {# #
74#FFT#75#0#Elevator *e_ptr;# #
75#FFT#77#0#e_ptr = new Elevator(10);# #
77#FFT#9#78#Elevator (int l_top_floor) {#e_ptr->go(2);#
9#FFT#10#0#current_floor = 1;# #
10#FFT#11#0#current_direction = UP;# #
11#FFT#12#0#top_floor = l_top_floor;# #
12#FFT#13#0#bottom_floor = 1;# #
13#FFT#0#0# # #
78#FFT#30#0#virtual void go (int floor) {# #
30#FFT#31#0#int valid = valid_floor(floor);# #
31#FFT#61#32#int valid_floor(floor) {#if (!valid) {#
61#FFT#62#0#if ((floor > top_floor) || (floor < bottom_floor))# #
62#FFT#63#64#return 0;#return 1;#
63#FFT#0#0# # #
64#FFT#0#0# # #
32#TTF#33#36#cout << "Invalid floor request\n";#if (floor > current_floor) {)#
33#FFT#34#0#return;# #
34#FFT#0#0# # #
36#TTF#37#40#up();#else if (floor <= current_floor) {#
37#FFT#18#38#void up() {#cout << "Elevator is going up";#
18#FFT#19#0#current_direction = UP;# #
19#FFT#0#0# # #
38#FFT#46#0#if (current_direction == UP) {# #
40#TTF#41#44#down();#cout << "Elevator is on the same floor\n";#
41#FFT#22#42#void down(){#cout << "Elevator is going down";#
22#FFT#23#0#current_direction = DOWN;# #
23#FFT#0#0# # #
42#FFT#46#0#if (current_direction == UP) {# #
46#TTF#47#51#while ((current_floor != floor) && (current_floor <= top_floor))#while
((current_floor != floor) && (current_floor <= bottom_floor))#
47#TTF#48#0#add(current_floor, 1);# #
48#FFT#57#0#add(int &a, const int &b) {# #
57#FFT#58#0#a = a+b;# #
58#FFT#0#0# # #
51#TTF#52#0#add(current_floor, -1)# #
52#FFT#0#0# # #
44#FFT#45#0#return;# #
45#FFT#0#0# # #
%
```

Figure 5.1. (f) Input file containing ICFG information for the modified version of the Elevator application.

test	top_floor	bottom_floor	Current_floor	Floor	edge trace
t1	10	1	1	-1	(74,75),(75,77),(77,9),(9,10),(10,11),(11,12),(12,13),(13,78),(78,30),(30,31),(31,61),(61,62),(62,63),(63,32),(32,33),(33,34)
t2	10	1	1	5	(74,75),(75,77),(77,9),(9,10),(10,11),(11,12),(12,13),(13,78),(78,30),(30,31),(31,61),(61,62),(62,64),(64,32),(32,36),(36,37),(37,18),(18,19),(19,38),(38,46),(46,47),(47,48),(48,57),(57,58),(58,47)
t3	10	1	5	2	(74,75),(75,77),(77,9),(9,10),(10,11),(11,12),(12,13),(13,78),(78,30),(30,31),(31,61),(61,62),(62,64),(64,32),(32,36),(36,40),(40,41),(41,22),(22,23),(23,42),(42,46),(46,51),(51,52),(52,57),(57,58),(58,51)
t4	10	1	2	2	(74,75),(75,77),(77,9),(9,10),(10,11),(11,12),(12,13),(13,78),(78,30),(30,31),(31,61),(61,62),(62,64),(64,32),(32,36),(36,40),(40,45)

Figure 5.1. (g) Test history information table for Elevator application.

	Modification traversing	
test	(36,40)	(40,45)
t1		
t2		
t3	X	
t4	X	X

Figure 5.1. (h) Test(s) selected to be reexecuted on the modified version of the Elevator application.

Figures 5.2 (a) to (h) contain the same information as figures 5.1 (a) to (h) but for the Time application. The time application is formed of one class which is the time class with two constructors and no destructor, two member functions and three data items. Two statements are altered: statement number 25 and statement number 30. Line 15 and 19 in figure 5.2 (f) differs from line 15 and 19 in figure 5.2 (c) concerning the statements equivalence. Four tests are executed on the time application. We assume that a test driver sets the values of t1.hrs, t1.mins, t1.secs, t2.hrs, t2.mins, and t3.secs. The table in figure 5.2 (h) is formed with two columns: the first column represents the result when the above mentioned modification are applied, two edges (24,25) and (29,30) are selected by the program, and three tests t1, t2, and t3 are selected according to these edges; the second column represents the result when only one statement is altered (30), only one edge (29,30) is selected and two tests t1 and t2 containing that edge are selected accordingly.

```
#include <iostream.h>

class time{
private:
   int hrs, mins, secs;

public:
   time(){
      hrs = mins = secs = 0;
   }

   time (int h, int m, int s) {
      hrs = h;
      mins = m;
      secs = s;
   }

   void display()     {
      cout << hrs << ":" << mins << ":" << secs;
   }

   void add_time(time t1, time t2){
      secs = t1.secs + t2.secs;
      if ( secs > 59 ) {
         secs -= 60;
         mins++;
      }
      mins += t1.mins + t2.mins;
      if ( mins > 59 ){
         mins -= 60;
         hrs++;
      }
hrs += t1.hrs + t2.hrs;
}
};

void main()             {
time time1(5, 59, 59);
time time2(4, 30, 30);
time time3;

time3.add_time(time1, time2);
cout << "\ntime3 = ";
time3.display();
}
```

Figure 5.2. (a) Actual code for the base version of the Time application.

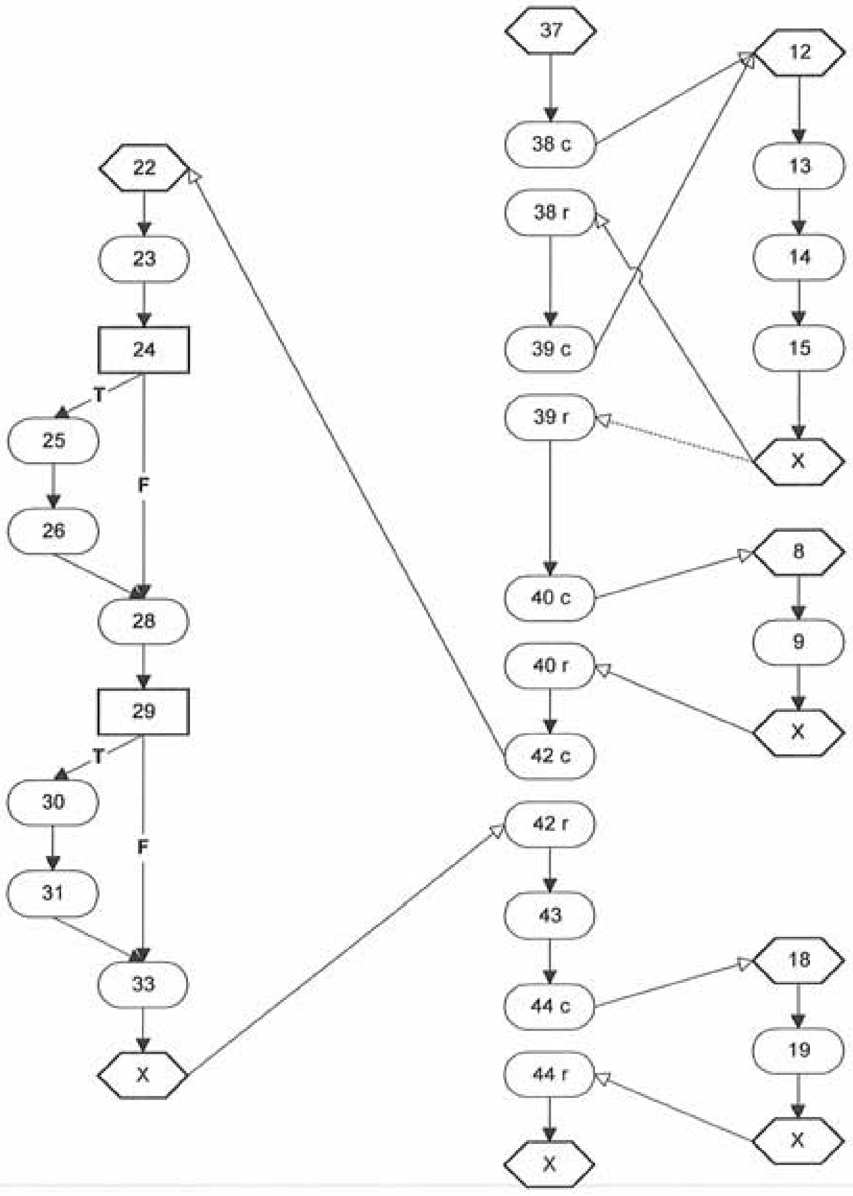

Figure 5.2. (b) ICFG for the base version of the Time application.

```
0#FFT#37#0#void main( ) {# #
37#FFT#38#0#time time1(5, 59, 59);# #
38#FFT#12#39#time (int h, int m, int s) {#time time2(4, 30, 30);#
12#FFT#13#0#hrs = h;# #
13#FFT#14#0#mins = m;# #
14#FFT#15#0#secs = s;# #
15#FFT#0#0# # #
39#FFT#12#40#time (int h, int m, int s) {#time time3;#
40#FFT#8#42#time( ) {#time3.add_time(time1, time2);#
8#FFT#9#0#hrs = mins = secs = 0;# #
9#FFT#0#0# # #
42#FFT#22#43#void add_time(time t1, time t2) {#cout << "\ntime3 = ";#
22#FFT#23#0#secs = t1.secs + t2.secs;# #
23#FFT#24#0#if ( secs > 59 ) {# #
24#TTF#25#28#secs -= 60;#mins += t1.mins + t2.mins;#
25#FFT#26#0#mins++;# #
26#FFT#28#0#mins += t1.mins + t2.mins;# #
28#FFT#29#0#if ( mins > 59 ) {# #
29#TTF#30#33#mins -= 60;#hrs += t1.hrs + t2.hrs;#
30#FFT#31#0#hrs++;# #
31#FFT#33#0#hrs += t1.hrs + t2.hrs;# #
33#FFT#0#0# # #
43#FFT#44#0#time3.display( );# #
44#FFT#18#0#void display( ){# #
18#FFT#19#0#cout << hrs << " : " << mins << " : " << secs;# #
19#FFT#0#0# # #
%
```

Figure 5.2. (c) Input file containing ICFG information for the base version of the Time application.

```
#include <iostream.h>

class time{
private:
int hrs, mins, secs;

public:
    time(){
        hrs = mins = secs = 0;
    }

    time (int h, int m, int s) {
        hrs = h;
        mins = m;
        secs = s;
    }

    void display()      {
        cout << hrs << ":" << mins << ":" << secs;
    }

    void add_time(time t1, time t2){
        secs = t1.secs + t2.secs;
        if ( secs > 59 ) {
            secs = secs - 60;
            mins++;
        }
        mins += t1.mins + t2.mins;
        if ( mins > 59 ){
            mins = mins - 60;
            hrs++;
        }
hrs += t1.hrs + t2.hrs;
}
};

void main()             {
time time1(5, 59, 59);
time time2(4, 30, 30);
time time3;

time3.add_time(time1, time2);
cout << "\ntime3 = ";
time3.display();
}
```

Figure 5.2. (d) Actual code for the modified version of the Time application.

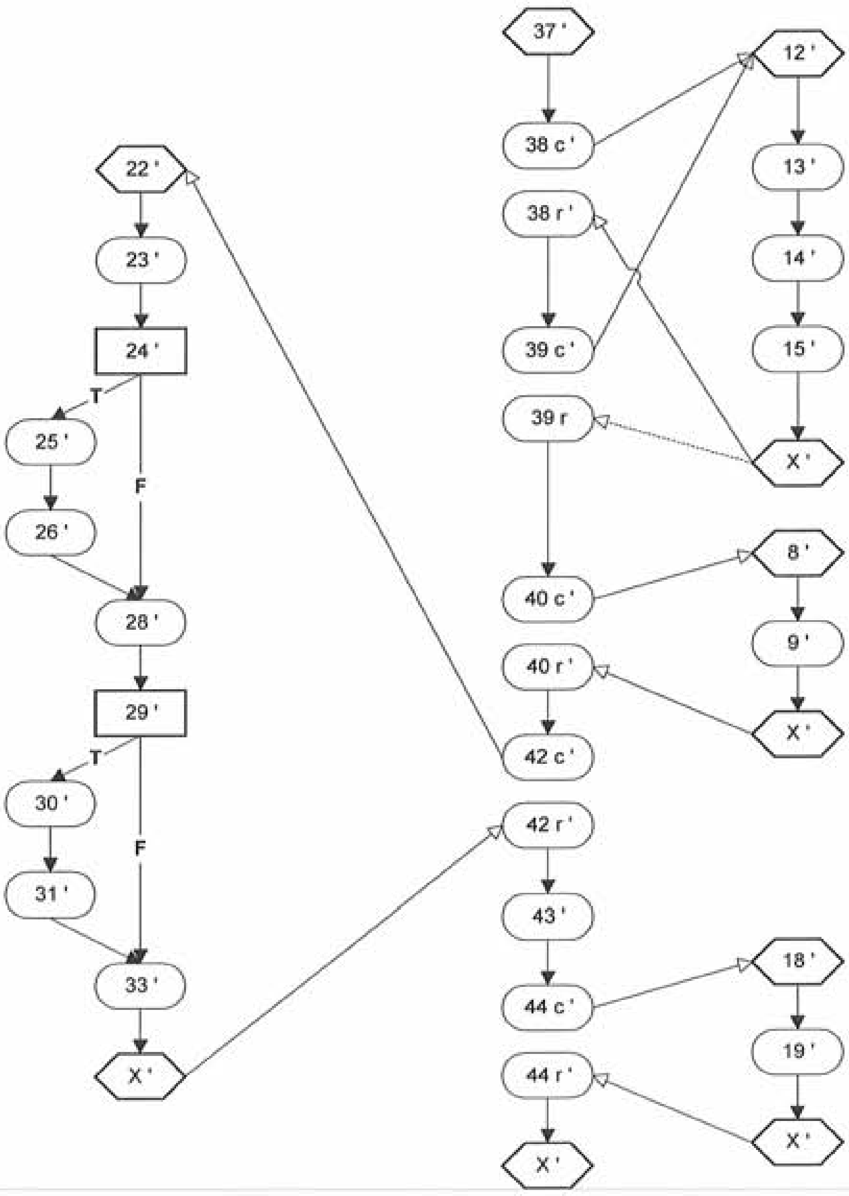

Figure 5.2. (e) ICFG for the modified version of the Time application.

```
0#FFT#37#0#void main( ) {# #
37#FFT#38#0#time time1(5, 59, 59);# #
38#FFT#12#39#time (int h, int m, int s) {#time time2(4, 30, 30);#
12#FFT#13#0#hrs = h;# #
13#FFT#14#0#mins = m;# #
14#FFT#15#0#secs = s;# #
15#FFT#0#0# # #
39#FFT#12#40#time (int h, int m, int s) {#time time3;#
40#FFT#8#42#time( ) {#time3.add_time(time1, time2);#
8#FFT#9#0#hrs = mins = secs = 0;# #
9#FFT#0#0# # #
42#FFT#22#43#void add_time(time t1, time t2) {#cout << "\ntime3 = ";#
22#FFT#23#0#secs = t1.secs + t2.secs;# #
23#FFT#24#0#if ( secs > 59 ) {# #
24#TTF#25#28#secs = secs-60;#mins += t1.mins + t2.mins;#
25#FFT#26#0#mins++;# #
26#FFT#28#0#mins += t1.mins + t2.mins;# #
28#FFT#29#0#if ( mins > 59 ) {# #
29#TTF#30#33#mins = mins-60;#hrs += t1.hrs + t2.hrs;#
30#FFT#31#0#hrs++;# #
31#FFT#33#0#hrs += t1.hrs + t2.hrs;# #
33#FFT#0#0# # #
43#FFT#44#0#time3.display( );# #
44#FFT#18#0#void display( ){# #
18#FFT#19#0#cout << hrs << " : " << mins << " : " << secs;# #
19#FFT#0#0# # #
%
```

Figure 5.2. (f) Input file containing ICFG information for the modified version of the Time application.

Test	t1.hrs	t1.mins	t1.secs	t2.hrs	t2.mins	t2.secs	edge trace
t1	5	59	59	4	30	30	(37,38),(38,12),(12,13),(13,14),(14,15),(15,39),(39,12),(15,40),(40,8),(8,9),(9,42),(42,22),(22,23),(23,24),(24,25),(25,26),(26,28),(28,29),(29,30),(30,31),(31,33),(33,43),(43,44),(44,18),(18,19)
t2	5	30	10	4	30	15	(37,38),(38,12),(12,13),(13,14),(14,15),(15,39),(39,12),(15,40),(40,8),(8,9),(9,42),(42,22),(22,23),(23,24),(24,28),(28,29),(29,30),(30,31),(31,33),(33,43),(43,44),(44,18),(18,19)
t3	5	10	30	4	15	30	(37,38),(38,12),(12,13),(13,14),(14,15),(15,39),(39,12),(15,40),(40,8),(8,9),(9,42),(42,22),(22,23),(23,24),(24,25),(25,26),(26,28),(28,29),(29,33),(33,43),(43,44),(44,18),(18,19)
t4	5	10	10	4	10	10	(37,38),(38,12),(12,13),(13,14),(14,15),(15,39),(39,12),(15,40),(40,8),(8,9),(9,42),(42,22),(22,23),(23,24),(24,28),(28,29),(29,33),(33,43),(43,44),(44,18),(18,19)

Figure 5.2. (g) Test history information table for Time application.

	Modification traversing	
test	(24,25), (29,30)	(29,30)
t1	X	X
t2	X	X
t3	X	
t4		

Figure 5.2. (h) Test(s) selected to be reexecuted on the modified version of the Time application.

Figures 5.3 (a) to (h) contain the same information as figures 5.1 (a) to (h) but for the publication application. The publication application is formed of two main classes and two other derived classes. The main classes are: the publication class with two member functions and two data items; and the sales class with two member functions and one data item. The derived classes are: the book class, which is derived from publication and sales classes, containing two member functions and one data item; and the tape class, which is derived from publication and sales classes, containing two member functions and one data items. One statement is altered: statement number 40 and two constructors are added: one for the derived class book, and the other for the derived class tape. Figure 5.3 (f) differs Figure 5.3 (c) concerning the statements equivalence and addition of a number of lines for the constructor of the two derived classes. Two tests are executed on the publication application. We assume that a test driver sets the values of Max, Months, and sales[0]. The table in figure 5.3 (h) is formed with two columns: the first column represents the result when the above mentioned modification are applied, one edge (90,91) is selected by the program, and the two tests t1 and t2 are selected according to this edge; the second column represents the result when only one statement is altered (40) and no constructors are added, only one edge (39,40) is selected and one test t1 containing that edge is selected accordingly.

```
#include <iostream.h>
const int LEN = 80;
const int MONTHS = 3;
const int MAX = 500;

class publicaion {
private:
  char title[LEN];
  float price;

public:
  void getdata()      {
    cout << "\nEnter title: ";
    cin >> title;
    cout << "Enter price: ";
    cin >> price;
  }

  void putdata()      {
    cout << "\nTitle: " << title;
    cout << "\n     Price: " << price;
  }
};

class sales {
private:
  float sales[MONTHS];

public:
  void getdata();
  void putdata();
};

void sales::getdata()   {
  cout << "   Enter sales for 3 months\n";
  for(int j=0; j<MONTHS; j++)
  {
     cout << "             Month " << j+1 << ": ";
     cin >> sales[j];
     if (sales[j] > MAX)
       sales[j] = MAX;
  }
}

void sales::putdata()   {
  for(int j=0; j<MONTHS; j++)
  {
     cout << "             Sales for month " << j+1 << ": ";
     cout >> sales[j];
  }
}

class book : private publication, private sales     {
private:
  int pages;

public:
  void getdata()      {
    publication::getdata();
    cout << "Enter number of pages: ";
    cin >> pages;
    sales::getdata();
  }

  void pudata(){
    publication::putdata();
    cout << "\n     Pages: " << pages;
    sales::putdata();
  }
};

class tape : private publication, private sales     {
private:
  float time;

public:
  void getdata()      {
    publication::getdata();
    cout << "Enter playing time: ";
    cin >> time;
    sales::getdata();
  }

  void pudata(){
    publication::putdata();
    cout << "\n     Playing time: " << time;
```

```
      sales::putdata( );
   }
};

void main( )             {
   book book1;
   tape tape1;

   book1.getdata( );
   tape1.getdata( );

   book1.putdata( );
   tape1.putdata( );
}
```

Figure 5.3. (a) Actual code for the base version of the Publication application.

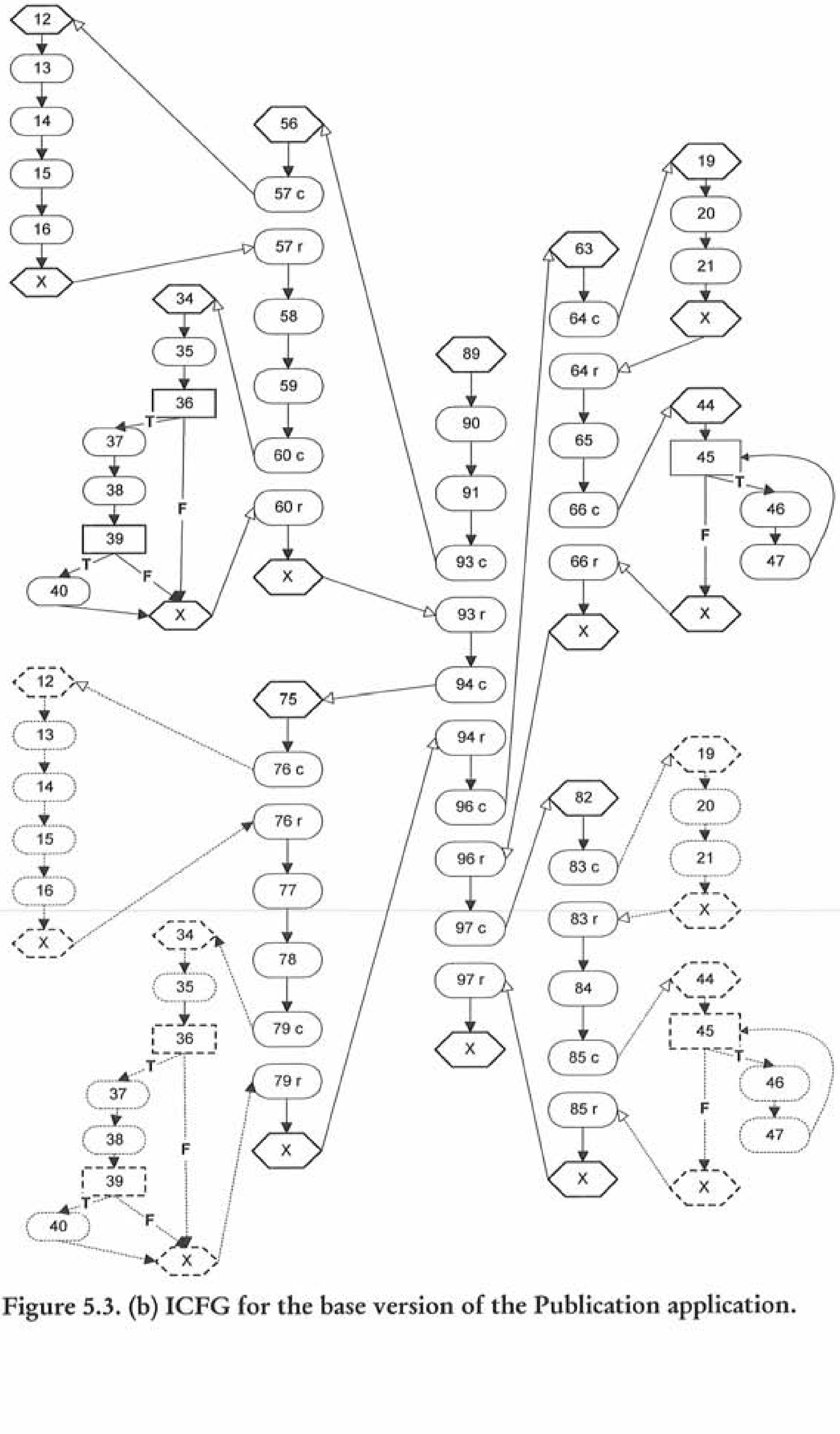

Figure 5.3. (b) ICFG for the base version of the Publication application.

```
0#FFT#89#0#void main( )    {# #
89#FFT#90#0#book book1;# #
90#FFT#91#0#tape tape1;# #
91#FFT#93#0#book1.getdata( );# #
93#FFT#56#94#void getdata( ) {#tape1.getdata( );#
56#FFT#57#0#publication::getdata( );# #
57#FFT#12#58#void getdata( ) {#cout << " Enter number of pages: ";#
12#FFT#13#0#cout << "\nEnter title: "; # #
13#FFT#14#0#cin >> title;# #
14#FFT#15#0#cout << "      Enter price: ";# #
15#FFT#16#0#cin >> price;# #
16#FFT#0#0# # #
58#FFT#59#0#cin >> pages;# #
59#FFT#60#0#sales::getdata( );# #
60#FFT#34#0#void sales::getdata() {# #
34#FFT#35#0#cout << "      Enter sales for 3 months\n";# #
35#FFT#36#0#for(int j=0; j<MONTHS; j++) {# #
36#TTF#37#0#cout << "  Month " << j+1 << ": ";# #
37#FFT#38#0#cin >> sales[j];# #
38#FFT#39#0#if (sales[j] > MAX)# #
39#TTF#40#0#sales[j]=MAX;# #
40#FFT#0#0# # #
94#FFT#75#96#void getdata( ) {#book1.putdata( );#
75#FFT#76#0#publication::getdata( );# #
76#FFT#12#77#void getdata( ) {#cout << " Enter playing time: ";#
77#FFT#78#0#cin >> time;# #
78#FFT#79#0#sales::getdata( );# #
79#FFT#34#0#void sales::getdata() {# #
96#FFT#60#97#void pudata( ) {#tape1.putdata( );#
63#FFT#64#0#publication::putdata( );# #
64#FFT#12#65#void getdata( ) {#cout << "\n Pages: " << pages;#
65#FFT#66#0#sales::putdata( );# #
66#FFT#34#0#void sales::getdata() {# #
97#FFT#82#0#void pudata( ) {# #
82#FFT#83#0#publication::putdata( );# #
83#FFT#12#84#void getdata( ) {#cout << "\n Playing time: " << time;#
84#FFT#85#0#sales::putdata( );# #
85#FFT#34#0#void sales::getdata() {# #
%
```

Figure 5.3. (c) Input file containing ICFG information for the base version of the Publication application.

```
1' #include <iostream.h>
2' const int LEN = 80;
3' const int MONTHS =3;
4' const int MAX = 500;
5'
6' class publicaion {
7' private:
8' char title[LEN];
9' float price;
10'
11' public:
12' void getdata()  {
13' cout << "\nEnter title: ";
14' cin >> title;
15' cout << "     Enter price: ";
16' cin >> price;
17' }
18'
19' void putdata() {
20' cout << "\nTitle: " << title;
21' cout << "\n   Price: " << price;
22' }
23' };
24'
25' class sales {
26' private:
27' float sales[MONTHS];
28'
29' public:
30' void getdata();
31' void putdata();
32' };
33'
34' void sales::getdata()  {
35' cout << "     Enter sales for 3 months\n";
36' for(int j=0; j<MONTHS; j++)  {
37' cout << "              Month " << j+1 << ": ";
38' cin >> sales[j];
39' if (sales[j] > MAX)
40' sales[j] = MAX + 1;
41' }
42' }
43'
44' void sales::putdata()  {
45' for(int j=0; j<MONTHS; j++)  {
46' cout << "              Sales for month " << j+1 << ": ";
47' cout >> sales[j];
48' }
49' }
50'
51' class book : private publication, private sales    {
52' private:
53' int pages;
54'
55' public:
55.1  book(){
55.2      pages = 0;}
56' void getdata()  {
57' publication::getdata();
58' cout << "     Enter number of pages: ";
59' cin >> pages;
60' sales::getdata();
61' }
62'
63' void pudata()  {
64' publication::putdata();
65' cout << "\n   Pages: " << pages;
66' sales::putdata();
67' }
68' };
69'
70' class tape : private publication, private sales    {
71' private:
72' float time;
73'
74' public:
74.1  tape() {
74.2  time = 0.0;}
75' void getdata()  {
76' publication::getdata();
77' cout << "     Enter playing time: ";
78' cin >> time;
79' sales::getdata();
80' }
81'
82' void pudata()  {
83' publication::putdata();
cout << "\n   Playing time: " <<
    time;
sales::putdata();
}
};

void main()          {
book book1;
tape tape1;

book1.getdata();
tape1.getdata();

book1.putdata();
tape1.putdata();
}

```

Figure 5.3. (d) Actual code for the modified version of the Publication application.

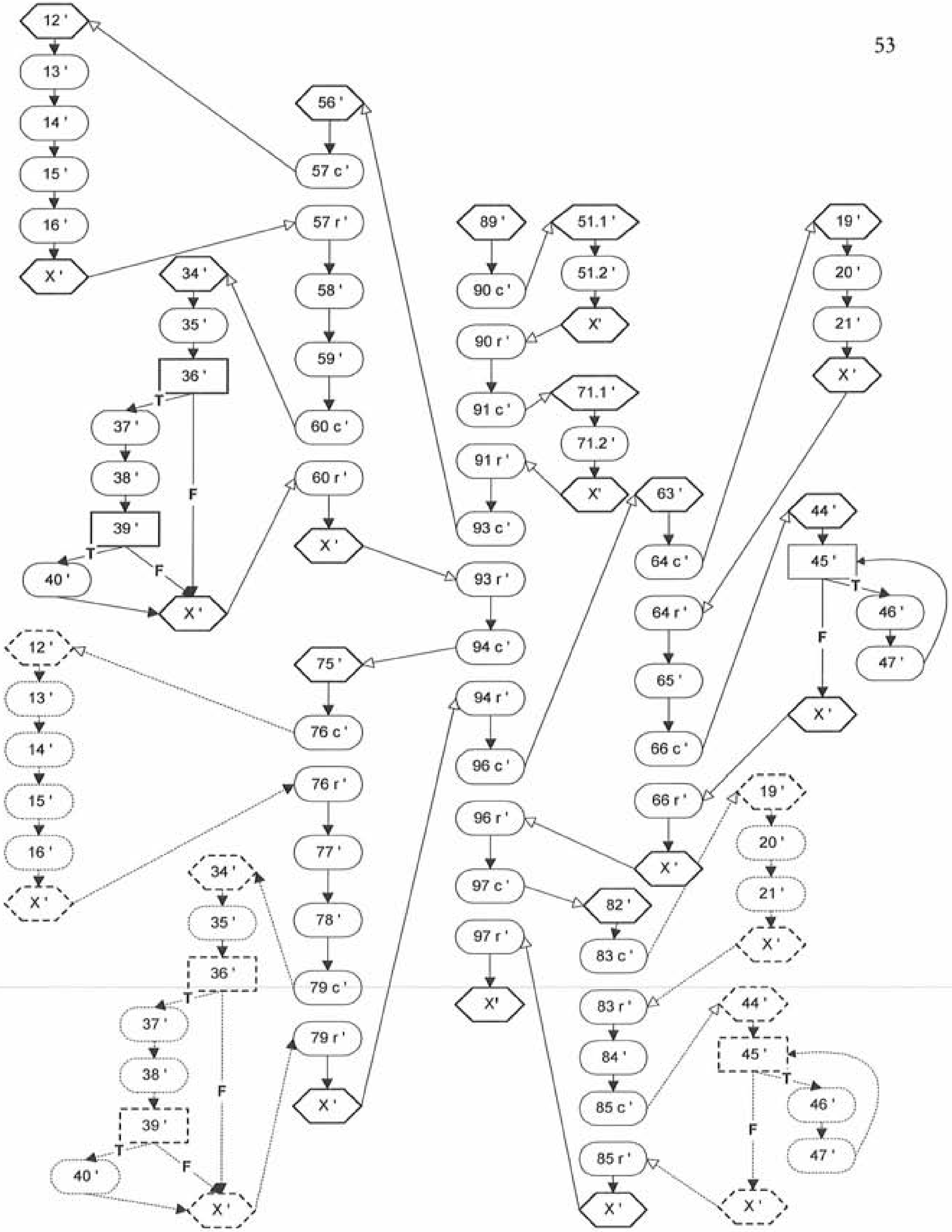

Figure 5.3. (e) ICFG for the modified version of the Publication application.

```
0#FFT#89#0#void main( )    {# #
89#FFT#90#54#book book1;#book( ) {#
54#FFT#55#0#pages = 0;# #
55#FFT#0#0# # #
90#FFT#91#73#tape tape1;#tape( ) {#
73#FFT#74#0#time = 0.0;# #
74#FFT#0#0# # #
91#FFT#93#0#book1.getdata( );# #
93#FFT#56#94#void getdata( ) {#tape1.getdata( );#
56#FFT#57#0#publication::getdata( );# #
57#FFT#12#58#void getdata( ) {#cout << " Enter number of pages: ";#
12#FFT#13#0#cout << "\nEnter title: "; # #
13#FFT#14#0#cin >> title;# #
14#FFT#15#0#cout << "       Enter price: ";# #
15#FFT#16#0#cin >> price;# #
16#FFT#0#0# # #
58#FFT#59#0#cin >> pages;# #
59#FFT#60#0#sales::getdata( );# #
60#FFT#34#0#void sales::getdata() {# #
34#FFT#35#0#cout << "       Enter sales for 3 months\n";# #
35#FFT#36#0#for(int j=0; j<MONTHS; j++) {# #
36#TTF#37#0#cout << "  Month " << j+1 << ": ";# #
37#FFT#38#0#cin >> sales[j];# #
38#FFT#39#0#if (sales[j] > MAX)# #
39#TTF#40#0#sales[j]=MAX+1;# #
40#FFT#0#0# # #
94#FFT#75#96#void getdata( ) {#book1.putdata( );#
75#FFT#76#0#publication::getdata( );# #
76#FFT#12#77#void getdata( ) {#cout << " Enter playing time: ";#
77#FFT#78#0#cin >> time;# #
78#FFT#79#0#sales::getdata( );# #
79#FFT#34#0#void sales::getdata() {# #
96#FFT#60#97#void pudata( ) {#tape1.putdata( );#
63#FFT#64#0#publication::putdata( );# #
64#FFT#12#65#void getdata( ) {#cout << "\n Pages: " << pages;#
65#FFT#66#0#sales::putdata( );# #
66#FFT#34#0#void sales::getdata() {# #
97#FFT#82#0#void pudata( ) {# #
82#FFT#83#0#publication::putdata( );# #
83#FFT#12#84#void getdata( ) {#cout << "\n Playing time: " << time;#
84#FFT#85#0#sales::putdata( );# #
85#FFT#34#0#void sales::getdata() {# #
%
```

Figure 5.3. (f) Input file containing ICFG information for the modified version of the Publication application.

test	MAX	MONTHS	Sales[0]	Edge trace
t1	500	1	501	(89,90),(90,91),(91,93),(93,56),(56,57),(57,12),(12,13),(13,14),(14,15),(15,16),(16,58),(58,59),(59,60),(60,34),(34,35),(35,36),(36,37),(37,38),(38,39),(39,40),(40,94),(94,75),(75,76),(76,12),(16,77),(77,78),(78,79),(79,34),(40,96),(96,63),(63,64),(64,19),(19,20),(20,21),(21,84),(84,85),(85,44)
t2	500	1	400	(89,90),(90,91),(91,93),(93,56),(56,57),(57,12),(12,13),(13,14),(14,15),(15,16),(16,58),(58,59),(59,60),(60,34),(34,35),(35,36),(36,37),(37,38),(38,39),(39,94),(94,75),(75,76),(76,12),(16,77),(77,78),(78,79),(79,34),(40,96),(96,63),(63,64),(64,19),(19,20),(20,21),(21,84),(84,85),(85,44)

Figure 5.3. (g) Test history information table for Publication application.

	Modification traversing	
test	(90,91)	(39,40)
t1	X	X
t2	X	

Figure 5.3. (h) Test(s) selected to be reexecuted on the modified version of the Publication application.

Chapter 6: Conclusion

The proposed technique is based on code analysis for selecting regression tests for C++ software. This technique selects test cases from existing test suites that execute code that has changed in the production of a C++ applications program or class. The results indicate that the proposed algorithm can reduce the number of regression tests that must be run. However, a reduction in the number of tests that must be run does not, by itself, promise savings; savings will result only if the cost of test selection analysis plus the cost of running the selected tests is less than the cost of simply running all tests. If test execution and validation are fully automated, and sufficiently efficient, then test selection may be unnecessary. If execution and/or validation are excessively expensive in time or human interaction, and if test selection analysis is sufficiently inexpensive, test selection may be worthwhile. The approach is advantageous because it handles selective retest needs for C++, and is independent of program specifications and methods used to develop test suites initially. Although this current work has addressed test selection for C++ software, the approach can be adapted to other strongly-typed languages such as Java.

Also of importance for future work is considering other important regression testing problems. It has been considered only the problem of selecting test cases, from an existing test suite, for reexecution. In general, modifications to classes may render existing test suite inadequate: new test cases to exercise new functionality are needed, and the problem of identifying where such test cases are needed is at least as important for software quality as the regression test selection problem.

Bibliography

[1] S. Elbaum, D. Gable, and G. Rothermel. Understanding and Measuring the Sources of Validations in the Prioritization of Regression Test Suites.

[2] P.Hsia, X. Li, D. Kung, C-T. Hsu, L Li, Y. Toyoshima, and C. Chen. A technique for the selective revalidation of OO software. *Software Maintenance: Research and Practice*, 9:217-233, 1997.

[3] D. Kung, J. Gao, and P. Hsia. Class Firewall, Test Order, and Regression Testing of Object-Oriented Programs. Department of Computer Science and Engineering. The University of Texas at Arlington.

[4] D. Kung, J. Gao, P.Hsia, F. Wen, Y. Toyoshima, and C. Chen. On regression testing of object-oriented programs. *The journal of Systems and Software*, 32(1):21-40, January 1996.

[5] L. Larsen and M.J. Harrold. Slicing object-oriented software. *In 18th International Coference on Software Engineering*, pages 495-505, March 1996.

[6] J. Hartmann and D.J. Robson. Techniques for selective revalidation. *IEEE Software*, 16(1):31-8, January 1990.

[7] Michelle Lee, A. Jefferson Offutt and Roger T. Alexander. Algorithmic Analysis of the Impacts of Changes to Object-oriented Software. *34th International Conference on Technology of Object-Oriented Languages and Systems*, Santa Barbara, CA, August 2000.

[8] H.K.N. Leung and L.White. Insights into testing and regression testing global variables. *Journal of Software Maintenance: Research and Practice*, 2:209-222, December 1990.

[9] H.K.N. Leung and L.White. A study of integration testing and software regression at the integration level. *In Proceedings of the Conference on Software Maintenance – 1990*, pages 290-300, November 1990.

[10] H.K.N. Leung and L.White. A cost model to compare regression test strategies. *In Proceedings of the Conference on Software Maintenance – 1991*, pages 201-8, October 1991.

[11] G. Rothermel and M.J. Harrold. A safe, efficient algorithm for regression test selection. *In Proceedings of the Conference on Software Maintenance – 1993*, pages 358-67, September, 1993.

[12] G. Rothermel and M.J. Harrold. Selecting regression tests for object-oriented software. *In Proceedings of the Conference on Software Maintenance – 1994*, pages 14-25. IEEE Computer Society Press, September 1994.

[13] G. Rothermel, M. J. Harrold, and J. Dedhia. Regression test selection for C++ software. *Journal of Software Testing, Verification and Reliability*, January 1999.

[14] M. Winter. Managing Object-Oriented Integration and Regression Testing. *In proceedings of the 6th European Conference on Software Testing Analysis Review*, Munich, Germany, Nov/Dec 1998.

[15] Lee J. White. Elements of Reuse and Regression Testing of Object-Oriented Software. Department of Computer Engineering and Science, Case Western Reverse University.

[16] L.J. White and H.K.N. Leung. A firewall concept for both control-flow and data-flow in regression integration testing. *In proceedings of the Conference on Software Maintenance, 1992*, pages 262-70, Novemeber 1992.

Appendix: Implementation Details

A. Creation of The Control Flow Graph

A full implementation, of the proposed technique, requires a front-end language analyzer that can instrument programs and output control flow graphs and symbol table information. Such an analyzer for C++ is not available, and therefore the creation of the graphs or test histories to be run on a C++ program cannot be accomplished. To compensate for the creation of a control flow graph and a history test table, input files are used instead to read from them similar information.

Two input files shall contain all needed information about the object-oriented program and its modified version, respectively. This information should be arranged in a way that imitates the role of the control flow graph and that, at the same time, can be used in the proposed technique. Another input file shall contain the test history of the object-oriented program by gathering edge trace information for each test belonging to the test suit created to test this program.

B. Files Format

The format of a line in the interprocedural control flow graph input file is:

- Node number
- Symbol "#"
- Three characters of Boolean type (T or F) representing the out edge label for true, false, normal edge.
- Symbol "#"
- First Successor node number

- Symbol "#"
- Second Successor node number
- Symbol "#"
- Actual code associated with the statement of the first successor node
- Symbol "#"
- Actual code associated with the statement of the second successor node
- Symbol "#"

The format of the line in the test history file is:

- Node number
- Symbol ","
- Node number
- Symbol #
- Node number
- Symbol ","
- Node number
- Symbol #
- Etc...

B. File Specifications and Rules

The rules that should be followed in order to create the control flow graph in the input file is the following:

- Every statement in the object-oriented program should be given a number starting from the first line of the program.

- Then create manually the interprocedural control flow graph (ICFG) of the program by starting with the entry node of the program, which is the main function.
- To simulate the ICFG, we used an input file formatted as described in the previous paragraph for both the original program and its modified version:
    - The first line in the input file should have a node number equal zero and the first successor node number is the node number of the program startup function, which is the main function.
    - A predicate node, which represents a conditional statement, should have two successors. Also, a call node also has two successors: the called entry node of the called method and the next statement that follows that call node. The exit node of a method has no successors.
    - A zero value, given to one of the two successor fields, means that the node being traversed has no successor.
    - The field, that contains the three Boolean characters, takes its value by studying the situation of the node being traversed. If the node is not a predicate then the first two characters have value FF, and the third T, meaning it is a normal statement. If the node is a predicate, then the first two characters have value TT, and the third F, meaning it is not a normal statement. In addition to that, the call node is treated like a normal node concerning these three characters.
    - At the end of each file a symbol "%" is added in the last line to denote the end of file.