An Approach to Regression Testing using Slicing†

Rajiv Gupta
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
gupta@cs.pitt.edu

Mary Jean Harrold
Department of Computer Science
Clemson University
Clemson, SC 29634-1906
harrold@cs.clemson.edu

Mary Lou Soffa
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
soffa@cs.pitt.edu

Abstract

After changes are made to a previously tested program, a goal of regression testing is to perform retesting based only on the modification while maintaining the same testing coverage as completely retesting the program. We present a novel approach to data flow based regression testing that uses slicing type algorithms to explicitly detect definition-use pairs that are affected by a program change. An important benefit of our slicing technique is, unlike previous techniques, no data flow history is needed nor is the recomputation of data flow for the entire program required to detect affected definition-use pairs. The program changes drive the recomputation of the required partial data flow through slicing. Another advantage is that our technique achieves the same testing coverage as a complete retest of the program without maintaining a test suite. Thus, the overhead of maintaining and updating a test suite is eliminated.

1. Introduction

Although software may be completely tested at some point during its development and maintenance, program changes require that parts of the software be retested. *Regression testing* is the process of validating the modified parts of the software and ensuring that no new errors are introduced into previously tested code. In addition to testing the changed code, regression testing must retest parts of the program affected by a change. A selective approach to regression testing attempts to identify and retest only those parts of the program that are affected by a change.

Techniques based on the data flow in a program have been developed for selective regression testing[6, 7, 10, 11]. In data flow testing[2, 3, 8, 9], a variable assignment at a point in a program is tested by generating test cases that execute subpaths from the assignment (i.e., *definition*) to points where the variable's value is used (i.e., *use*). Traditional data flow analysis techniques are used to compute definition-use (def-use) pairs, and test data adequacy criteria are used to select particular def-use pairs to test. Test cases are then generated that cause execution through selected def-use pairs. When a test case executes a definition and a use, that def-use pair is "satisfied". Selective regression testing techniques for data flow testing first identify the def-use pairs that are affected by a change and then test cases that satisfy these affected def-use pairs are run. Current techniques use test cases from a test suite along with newly generated test cases to satisfy the affected def-use pairs. The results of the executions are used to update the test suite.

Existing data flow based regression testing techniques explicitly identify *directly* affected def-use pairs by detecting *new* def-use pairs that are created by a program change. These techniques compute the changed data flow by either incrementally updating the original data flow to agree with the modified code[6, 7, 11] or exhaustively computing the data flow for both the original and modified programs and comparing the sets to determine the differences[13]. Thus, data flow information is either saved between testing sessions or completely recomputed at the beginning of each session. A program change can also *indirectly* affect def-use pairs due to either a change in the computed value of a definition or a change in the predicate value of a conditional statement. Current regression testing techniques do not explicitly identify these indirectly affected def-use pairs. Instead, indirectly affected def-use pairs are tested by running all test cases from the test suite that previously executed through the changed code although many of them are unnecessary.

This paper presents a new approach to selective regression testing using the concept of a **program slice**[12]. A program slice consists of all statements, including conditionals in the program, that might affect

† Partially supported by the National Science Foundation through Presidential Young Investigator Award CCR-9157371(9249143) and Grant CCR-9109089 to the University of Pittsburgh, and Grant CCR-9109531 to Clemson University.

0-8186-2980-0/92 $03.00 © 1992 IEEE

the value of variable V at point p. We use slicing algorithms to identify all def-use pairs that may be *directly* or *indirectly* affected by a program change. Our algorithms detect these def-use pairs without requiring either the data flow history or complete recomputation of data flow for the program. By explicitly identifying all affected def-use pairs, we do not require a test suite to enable selective retesting. If a test suite is maintained, we execute fewer test cases since only those test cases that may test affected def-use pairs are rerun.

Our technique uses two slicing type analysis algorithms to determine directly the affected def-use pairs (new pairs) and the indirectly affected def-use pairs. The first algorithm is a backward walk through the program from the point of the edit that searches for definitions related to the changed statement. The second algorithm is a forward walk from the point of the edit. During the forward walk, the algorithm detects the uses and the subsequent definitions and uses that are affected by a definition that is changed as a result of the program edit. Additionally, any def-use pairs that depend on a changed predicate are identified. Through these two walks, our algorithms detect the changed and affected def-use pairs due to program modifications.

The next section presents our analysis to detect affected def-use pairs and includes the algorithms for backward and forward walks. Section 2 also presents our update actions for different kinds of program edits. In Section 3, we discuss the merit of our technique, its applications to testing and the efficiency of our algorithms. Concluding remarks are given in Section 4.

2. Detecting Affected Definition-Use Pairs

Numerous data flow testing techniques [3, 8, 9] have been developed to assist in detecting program errors. All of these techniques use the data flow information in a program to guide the selection of test data. Traditional data flow analysis techniques [1] are used to compute the def-use pairs. A program is represented by a *control flow graph* where each node in the graph corresponds to a statement and each edge represents the flow of control between statements. Definitions and uses of variables are attached to nodes in the control flow graph and data flow analysis is performed to compute the def-use pairs. Uses are classified as either *computation* uses (c-uses) or *predicate* uses (p-uses). A c-use occurs whenever a variable is used in a computation statement; a p-use occurs whenever a variable is used in a conditional statement. In Figure 2, node 7 contains a c-use of the definition of X in node 4. We represent this def-use by the pair (4,7). Figure 2 also contains a p-use in node 5 of the definition of A in node 3. For testing purposes, there is a def-use pair from the definition in node 3 to each of the edges leaving the conditional node 5. Thus, we get the two def-use pairs (3,(5,6)) and (3,(5,7)).

Test data adequacy criteria are used to select particular def-use pairs/subpaths to test. One criterion, 'all-uses', requires that each definition of a variable be tested on some subpath to each of its uses. This testing criterion has been shown practical, since relatively few test cases are typically required for its satisfaction[14].

To satisfy the 'all-uses' data flow testing criterion after making a program change, we must identify the def-use pairs affected by the change. Affected def-use pairs fall into two categories: (1) those affected *directly* because of the insertion/deletion of definitions and uses (*new pairs*) and (2) those affected *indirectly* because of a change in either a computed value (*value pairs*) or a path condition (*path pairs*).

New Pairs. A program edit can cause creation of new def-use pairs that must be tested. For example, replacing statement "X:=2" with "X:=2+Y" introduces a new use of variable Y. Def-use pairs consisting of those definitions of Y that reach the new use of Y must be tested.

Value Pairs. Value pairs are the def-use pairs whose computed values may have changed because of the program edit and therefore, require retesting. For example, if a statement "X:=2" is changed to "X:=3", then although no new def-use pairs are created, we retest the def-use pairs that depend on the new value of X.

Path Pairs. The def-use pairs that are tested on a path whose path condition has changed must be retested. For example, if a branch condition "X<Y" is replaced by "X>Y", we must retest def-use pairs that were previously tested through the affected predicate. If the execution of the test case for a def-use pair encounters the definition prior to reaching the affected predicate and it encounters the use after executing the predicate, then we retest the def-use pair. In addition, the def-use pairs that are control dependent on the changed predicate must also be retested. A predicate can be affected by an explicit change to the predicate statement or by a program change that affects a value used in the predicate. Paths can also be affected by the deletion of an edge in the control flow graph.

In this section, we first discuss slicing algorithms for backward/forward walks that identify the definitions and uses that are affected by a program edit. Next, we consider different types of program edits and specify the actions required to identify the def-use pairs that must be tested to satisfy the 'all-uses' data flow testing criterion.

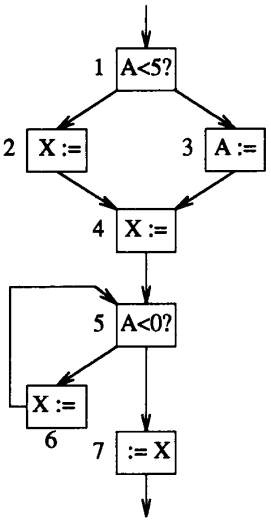
```
algorithm BackwardWalk(s,U)
input       s : program point/statement
            U : set of variables
output      DefsOfU : set of statements/nodes in the control flow graph corresponding to definitions
                      of variables in U that reach point s
declare     Worklist : statements/nodes in the control flow graph maintained as a priority queue
            In[i], Out[i], NewOut: set of program variables
            n, n_i : statement/node
            Pred(i), Succ(i) : returns the set of immediate predecessors(successors) of i
begin
    DefsOfU = Worklist = ∅                                   /* initialization */
    forall n ε Pred(s) do Worklist = n +(reverse-depth-first) Worklist
    In[S] = U;  Out[s] = ∅
    forall n_i ≠ s do In[n_i] = Out[n_i] = ∅
    while Worklist ≠ ∅ do                                    /* continue processing nodes */
        Get n from head of Worklist
        NewOut = ⋃(p ε Succ(n)) In[p]
        if NewOut ≠ Out[n] then                              /* change from last iteration */
            Out[n] = NewOut                                  /* recompute Out[n]*/
            if n defines a variable u ε U then               /* definition is found */
                DefsOfU = DefsOfU ⋃ {n}                      /* add n to definitions set */
                In[n] = In[n] - {u}                          /* stop searching for definitions of u */
            else In[n] = Out[n]                              /* no definition in n , just propagate */
            if In[n] ≠ ∅ then                                /*add Preds(n) to Worklist */
                forall x ε Preds(n) do Worklist = x +(reverse-depth-first) Worklist
    return(DefsOfU)
end BackwardWalk
```

Figure 1. Algorithm *BackwardWalk* computes the definitions of variables in U that reach the statement s.

2.1. Backward and Forward Walk Algorithms

Algorithms for backward and forward walks identify the definitions and uses that are affected by a program edit. Both algorithms use the control flow graph representation of the program where each node represents a single statement. These algorithms compute the data flow information to identify the affected def-use pairs but require no past history of data flow information. Furthermore, the algorithms are slicing algorithms in that they examine only the relevant parts of the control flow graph to compute the required data flow information. In our discussion, we assume that only scalars are being considered; the technique is easily extended to include arrays by adding a new condition for halting the search along paths.

The backward walk algorithm, *BackwardWalk* given in Figure 1, identifies the definitions of a set of variables U that reach a program point s. *BackwardWalk* traverses the control flow graph from the point s in the backward direction until definitions of all variables of U are encountered along each path. To assist in this process, sets of variables, *In* and *Out*, are maintained for relevant nodes in the control flow graph. *Out*[n] contains the variables whose definitions have not been encountered along some path from just after n to s; *In*[n] contains the variables whose definitions have not been encountered along some path from the point just before n to s. Since the algorithm walks backwards in the control flow graph, *Out*[n] is computed by taking the union of the *In* sets of n's successors. During this traversal, a worklist *Worklist*, consisting of those nodes that must be visited, is maintained to indicate how far the traversal has progressed. *Worklist* is maintained as a priority queue based on a reverse depth first ordering of nodes in the control flow graph. The statements in *Worklist* are examined for definitions of variables in U. As the statements in *Worklist* are examined, additional statements to be considered are determined and added to *Worklist*. Each statement n added to *Worklist* represents a point in the program along which the backward traversal must continue, since definitions of all variables in U were not encountered along a path from a successor of n to point s. Thus, n is only added to *Worklist* if the *In* set of n's successor is not empty. When the *Worklist* is empty all definitions of all variables in U have been encountered



BackwardWalk(7,{X})

defs of X visited	*path taken*
4	7,5,4
6	7,5,6

variable use	*def-use pairs*
use of X at statement 7	(6,7), (4,7)

Figure 2. *BackwardWalk* on variable X at statement 7. *BackwardWalk* visits definitions of X in nodes 4 and 6 and detects affected def-use pairs: (6,7) and (4,7).

along all backward paths from s and the algorithm terminates.

The example in Figure 2 demonstrates the application of *BackwardWalk* to locate the definitions of X that reach statement 7. *Worklist* is first initialized to statement 5, the only immediate predecessor of 7. After computing *In*[5] and *Out*[5], the immediate predecessors of statement 5, statements 4 and 6 are examined. Since statements 4 and 6 define variable X, the traversal stops and *BackwardWalk* returns statements 4 and 6.

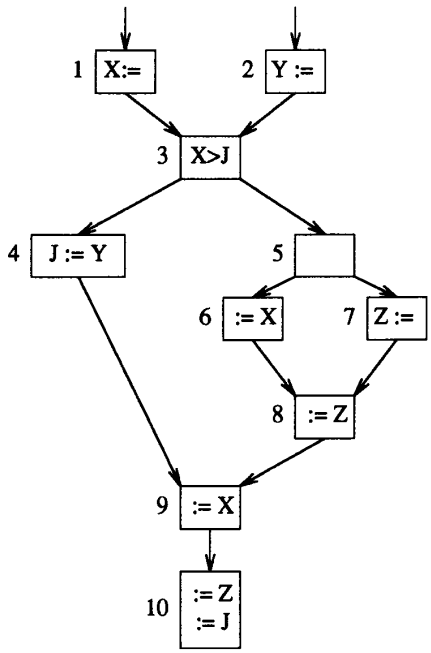
The forward walk algorithm, *ForwardWalk* given in Figure 3, identifies the uses of values, *ValueUseTriples*, that are directly or indirectly affected by a change in either a value of v at point s or a change in a predicate. *ValueUseTriples* have the form of triples (s, u, v) indicating that the value of variable v at statement s, affected by the change, is used by statement u. The returned *ValueUseTriples* are used to compute the affected def-use pairs. A def-use pair is directly affected if the pair represents a use of an altered definition. A def-use pair is indirectly affected in one of two ways: (1) the def-use pair is in the transitive closure of the changed definition or (2) the def-use pair is control dependent on a changed or affected predicate. *ForwardWalk* can handle multiple pairs, consisting of statement points and variables of the form (s,v), by simultaneously processing all these pairs. To simplify the discussion, we describe *ForwardWalk* for a single pair (s,v).

ForwardWalk inputs a set of *Pairs* representing the definitions whose uses will be found, along with a boolean, *PredOnly*, which indicates whether the walk starts with a set of definitions or a set of variable names at a program point. *PredOnly* is true if the walk begins at a point P and (s_i, v_i) is a definition s_i of variables v_i that reaches P and *Pairs* consists of (P, v_i). Otherwise, the walk begins with the pairs of affected definitions (s_i, v_i). For each statement node n, *In* and *Out* sets contain the definitions whose uses are to be found, since their values are affected by the edit. The set *In*[n] (*Out*[n]) contains the values just before (after) n whose uses are to be found. Each value is represented as a pair (d,p) indicating that the value of variable p at point d is of interest. If a statement n, that uses the value (d,p) belonging to *In*[n], is encountered, the def-use pair (d,n) is added to the list of def-use pairs affected by a change in the value of variable v at statement s. The value of the variable (say p') defined by the statement n is also indirectly affected. If a new definition of a variable p is encountered at statement n, then the values of p belonging to *In*[n] are killed by this definition and the search for these values along this path terminates. The set *Kill* in the algorithm denotes the set of values killed by a definition. The *Kill* set is needed to compute *Out*[n] from *In*[n]. Since the algorithm walks forward in the control flow graph, *In*[n] is computed by taking the union of the *Out* sets of n'n predecessors. During this traversal, a worklist, *Worklist*, consisting of those nodes that must be

```
algorithm ForwardWalk(Pairs, PredOnly)
input     Pairs: sets of definitions, (s_i, v_i), where s_i is a program point/statement and v_i is a variable
          PredOnly : boolean is true if change is only a predicate change
output    ValueUseTriples : (point/statement, statement, variable)
declare   In[i], Out[i], Kill, NewIn : set of pairs, (point/statement,variable)
          Worklist, Cd[i], OldCd, AffectedPreds : set of point/statement
          v : program variable
          k, n : statement/node
          DefsOfV[i] : set of (s, v) of definitions
          Pred(i), Succ(i) : returns the predecessors(successors) of i in the control flow graph
          Def(i): returns the variable defined by statement i
begin
    ValueUseTriples = ∅                                              /* initialization */
    forall (s,v) ε Pairs do
        forall n ε Succ(s) do Worklist = n +(depth-first) Worklist
    forall statements n_i not in any pair in Pairs do In[n_i] = Out[n_i] = ∅
    forall (s, v) ε Pairs do In[s] = ∅; Out[s] = {(s, v)}
    if PredOnly then AffectedPreds = {s_i} else AffectedPreds = ∅
    while Worklist ≠ ∅ do                                            /* continue processing nodes*/
        Get n from head of Worklist
        NewIn = ∪(p ε Pred(n)) Out[p]
        if NewIn ≠ In[n] then                                        /* change from last iteration */
                                                                     /* recompute In[n] */
            In[n] = NewIn
            OldCd = ∪(p ε Pred(n)) Cd(p)
            if OldCd - Cd(n) ≠ ∅ then
                forall k ε (OldCd - Cd(n)) ∩ AffectedPreds do
                    In[n] = In[n] - {(k, v_i) for all v_i}
                    AffectedPreds = AffectedPreds - {k}
                    forall (k, u, v) in ValueUseTriples do
                        ValueUseTriples = ValueUseTriples - {(k, u, v)}
                        forall (d, v) ε DefsOfV[k] do ValueUseTriples = ValueUseTriples ∪ {(d, u, v)}
            if n has a c-use of variable v such that (d,v) ε In[n] then   /* found a def-c-use pair */
                forall (d,v) ε In[n] do ValueUseTriples = ValueUseTriples ∪ {(d,n,p)}
                Kill = {(x,Def(n)): (x,Def(n)) ε In[n]}              /* propagate */
                Out[n] = (In[n] - Kill) ∪ {(n,Def(n))}
            elsif n has a p-use of variable v such that (d,v) ε In[n] then   /* found a def-p-use pair */
                forall (d,v) ε In[n] do ValueUseTriples = ValueUseTriples ∪ {(d,n,p)}
                DefsOfV[n] = BackwardWalk(n, {v}) - In[n]
                In[n] = In[n] ∪ {(n, v_i) : (d, v_i) ε DefsOfV(n)}
                Out[n] = In[n]
                AffectedPreds = AffectedPreds ∪ {n}
            elseif n defines a variable p and Cd(n) ∩ AffectedPreds ≠ ∅ then Out[n] = Out[n] ∪ {(n, Def(n))}
            else Out[n] = In[n]
            if Out[n] ≠ ∅ then
                forall x ε Succ(n) do Worklist = x +(depth-first) Worklist
    return(ValueUseTriples)
end ForwardWalk
```

Figure 3. Algorithm *ForwardWalk* identifies all value-use triples that are affected by a change in the values of each variable v_i at point s_i in the program or that are affected by a predicate change.

visited is maintained to indicate how far the traversal has progressed. *Worklist* is maintained as a priority queue based on a depth first ordering of nodes in the control flow graph. The statements in *Worklist* are examined for c-uses and p-uses of the definitions in the *In* sets along with definitions in statements that are control dependent



ForwardWalk({(2,Y)}, false)

variable	*def-use pair*	*path taken*
Y	(2,4)	2,3,4
J	(4,10)	4,9,10

ForwardWalk({(3,X),(3,Y)}, true)

variable	*def-use pair*	*path taken*
X	(1,6)	3,5,6
Y	(2,4)	2,3,4
J	(4,10)	4,9,10
Z	(7,8)	7,8
	(7,10)	7,8,9,10

Figure 4. The first *ForwardWalk* on variable Y at statement 2 finds def-use pairs (2,4) for Y and def-use pair (4,10) for J. The second *ForwardWalk* begins traversal at statement 3 with the definitions that reach it, (1,X) and (2,Y); def-use pairs (1,6) for X, (2,4) for Y and (4,10) for J are located. Since statement 7 is control dependent on statement 3, def-use pairs (7,8) and (7,10) are identified for Z.

on a changed or affected predicate. As the statements in *Worklist* are examined, additional statements to be considered are added to *Worklist*.

An important feature of *ForwardWalk* is that it identifies def-use pairs that are control dependent on an affected predicate, even though the value computed by the definition is unaffected. Thus, we require that the control dependency information is computed prior to using the *ForwardWalk* algorithm. Control dependencies are efficiently computed for each node in the control flow graph using the post-dominator relation among the nodes. For each control flow graph node n, $Cd[n]$ stores the set of nodes on which n is control dependent. A set of nodes *AffectedPreds* stores the current list of affected predicates. When a node containing a definition is encountered, if *AffectedPreds* is nonempty, the definition is added to the *In* set so that its uses can be found. In this way, *ForwardWalk* locates def-use pairs that are control dependent on affected predicates. *ForwardWalk* must also recognize when all def-use pairs that are control dependent on affected predicates are found. To accomplish this, each time a new node is removed from *Worklist*, its control dependencies are compared with the control dependencies of its predecessors. A decrease in control dependencies indicates that the predicate is no longer affected and it is removed from *AffectedPreds* and the *ValueUseTriples* are adjusted accordingly.

The example in Figure 4 illustrates the use of *ForwardWalk*. The first *ForwardWalk* on variable Y at statement 2 finds the use of Y in statement 4 to get the def-use pair (2,4). Since definition J in statement 4 is affected by the change in Y, *ForwardWalk* adds (4,J) to OUT[4] and continues traversal to find the def-use pair (4,10). In the second *ForwardWalk*, traversal begins with statement 3 and the definitions that reach it, (1,X) and (2,Y). Thus, the initial *Pairs* consist of (3,X) and (3,Y). For (3,X), *ForwardWalk* finds the *ValueUseTriple* (3, 6, X) that is used to identify the affected def-use pair (1,6) for X. However, def-use pair (1,9) is not included since its value is not affected by a change in statement 3. For (2,Y), *ForwardWalk* finds (2,4); since the definition of J in statement 4 is affected, (4,10) is identified. Since statement 7 is control dependent on statement 3, def-use pairs (7,8) and (7,10) are identified for Z even though their values are not affected by the change in statement 3.

2.2. Actions for Different Types of Edits

We now consider the way in which we use *BackwardWalk* and *ForwardWalk* to identify the def-use pairs that are affected by a program change. Instead of listing the entire algorithm for processing changes, we show the actions that are taken for each different type of edit. Each action results in a set, DefUsePairs, consisting of the affected def-use pairs.

Insert a use of variable X in statement S:Y=..X..

First, we identify and store in DefsOfX all definitions of X that reach the new use of X in S using *BackwardWalk*. Each of these definitions together with the use of X in the edit represents a newly created def-use pair that is added to NewPairs. *ForwardWalk* computes Triples that are the def-use pairs affected by the change. Using these Triples, we compute DefUsePairs consisting of those def-use pairs that have experienced value changes because of the change in the value of Y at S or def-use pairs that are control dependent on an affected predicate.

action InsertUse(S,X)
  DefsOfX = BackwardWalk (S,{X})
  NewPairs = $\bigcup_{S_i \varepsilon DefsOfX} \{(S_i,S)\}$
  Triples = ForwardWalk({(S,Y)}, false)
  DefUsePairs = {(def,use):(def,use,variable) ε Triples}
end InsertUse

Delete a use of variable X in statement S:Y=..X..

Deleting a use of X causes the value of Y to change. *ForwardWalk* identifies all value pairs that are affected by the change in the value of Y. If these def-use pairs involve conditions, then the path pairs for those conditions are also identified during *ForwardWalk*.

action DeleteUse(S,X)
  Triples = ForwardWalk({(S,Y)}, false)
  DefUsePairs = {(def,use):(def,use,variable) ε Triples}
end DeleteUse

Change operator in statement S:Y=..

If the operator used in an assignment statement is changed, no new def-use pairs are created. However, the value of the variable defined by this statement changes, which causes value pairs and possibly path pairs. *ForwardWalk* identifies both value pairs and path pairs affected by this change.

action OperatorChange(S)
  Triples = ForwardWalk({(S,Y)}, false)
  DefUsePairs = {(def,use):(def,use,variable) ε Triples}
end OperatorChange

Insert a definition of variable X in statement S:X=..

ForwardWalk identifies the newly created def-use pairs for the inserted definition of X. Other definitions that experience value changes because of the change in S are also found. If any of the def-use pairs involve predicates, the def-use pairs affected by those predicates are also identified during *ForwardWalk*.

action InsertDefinition(S,X)
  Triples = ForwardWalk({(S,X)}, false)
  DefUsePairs = {(def,use):(def,use,variable) ε Triples}
end InsertDefinition

Delete a definition of variable X in statement S:X=..

First, *BackwardWalk* identifies the definitions of X that reach the deleted definition of X and adds them to DefsOfX . Then, *ForwardWalk* identifies the new def-use pairs that are created by the edit at statement S, the value pairs whose values are affected, and def-use pairs dependent on affected predicates. Each value pair for X at S and any reachable use of this X is returned by *ForwardWalk*. Since we are deleting this definition of X, the definition in these def-use pairs must be replaced with the definitions of X found by *BackwardWalk* to determine the newly created def-use pairs.

action DeleteDefinition(S,X)
  DefsOfX = BackwardWalk (S,{X})
  Triples = ForwardWalk ({(S,X)}, false)
  NewPairs = {(S_i,S'): (S,S',X) ε Triples and S_i ε DefsOfX }
  ValuePathPairs = {(d,u): (d,u,V) ε Triples and d $\neq$ S}
DefUsePairs = NewPairs $\cup$ ValuePathPairs
end DeleteDefinition

Change operators in a branch condition C

Changing the operators of a branch condition (C) creates no new def-use pairs. Since the branch condition does not define any variable it does not directly affect the values of any def-use pairs. An example is changing a condition "X<Y" to "X>Y". However, we must identify all def-use pairs influenced by the result of C. These pairs are the value pairs and path pairs corresponding to C. The definitions that reach C are identified using *BackwardWalk*. The uses of these definitions that are reachable from C and are control dependent on C are identified using *ForwardWalk* (InPairs). *ForwardWalk* identifies any value pairs affected by changed values. *ForwardWalk* is also used to locate def-use pairs that are control dependent on C but whose values are not affected (ValuePathPairs).

```
action ChangeCondition(C)
  V = set of program variables
  DefsOfV = BackwardWalk (C,{V})
  Triples = ForwardWalk({(C,V_i): DefsOfV contains a
            definition of V_i}, true )
  InPairs = {(S_i,S'): (C,S',V) ε Triples and S_i ε DefsOfV }
  ValuePathPairs = {(d,u): (d,u,v) ε Triples and d ≠ C}
end ChangeCondition
```

Insert an edge e from statement p to statement t

Inserting an edge can cause creation of new def-use pairs which must be identified. The variables defined by the statements corresponding to the uses of the new def-use pairs have their values affected. Thus, all def-use pairs that depend upon the values of the affected variables must be identified. We first identify the definitions (DefsOfV) that reach t along the inserted edge by a backward walk. By performing forward walks from the points of these definitions, on the original flow graph (CFG), the def-use pairs (BeforePairs) involving the definitions that were originally present are identified. By performing forward walks form point p on the modified flow graph (CFG'), the def-use pairs (AfterPairs) involving the definitions that are present after the program change are identified. The difference of AfterPairs and BeforePairs gives us the newly created def-use pairs NewPairs. Next, the def-use pairs (ValuePathPairs) whose values are affected by the newly created def-use pairs are identified by a forward walk. The def-use pairs in NewPairs and ValuePathPairs are added to DefUse-Pairs.

```
action InsertEdge(e = p->t)
  V = set of program variables
  CFG is the original control flow graph
  CFG' is the modified control flow graph which includes
       edge e = p->t
  DefsOfV = BackwardWalk (t,{V}) on CFG'
  BeforeTriples = ForwardWalk ({(s,Def(s)): s ε DefsOfV},false )
       on CFG
  BeforePairs = {(def,use): (def,use,variable) ε BeforeTriples}
  AfterTriples = ForwardWalk ({(t,V_i): DefsOfV contains a definition
       of V_i and V_i ≠ Def(t)}, true ) on CFG'
  AfterPairs = {(s,t): s ε DefsOfV, Def(s) ε ref(t)} ∪
       {(s_i, s) : s_i ε DefsOfV and (p,s,V_i) ε AfterTriples}
  NewPathPairs = AfterPairs - BeforePairs
  ValueTriples = ForwardWalk({(s,Def(s)): oppE (r,s) ε NewPairs },
       false ) on CFG'
  ValuePathPairs = {(def,use): (def,use,variable) ε VALUE_Triples}
  DefUsePairs = NewPathPairs ∪ ValuePathPairs
end InsertEdge
```

Delete an edge e from statement p to statement t

Deleting an edge may cause the deletion of some def-use pairs but it does not create any new pairs. However, there are def-use pairs that are affected by the edge. An affected def-use pair is one that has not been deleted although the use was reachable by the definition through the deleted edge. First the definitions that reach statement s are identified by a backward walk. Next, using a forward walk from point p, all def-use pairs (BeforePairs) involving the above definitions and their uses reachable through the deleted edge are identified. The def-use pairs (DeletedPairs) that have been eliminated are identified and removed from BeforePairs, giving us DefUsePairs which are the def-use pairs in the modified flow graph which are affected.

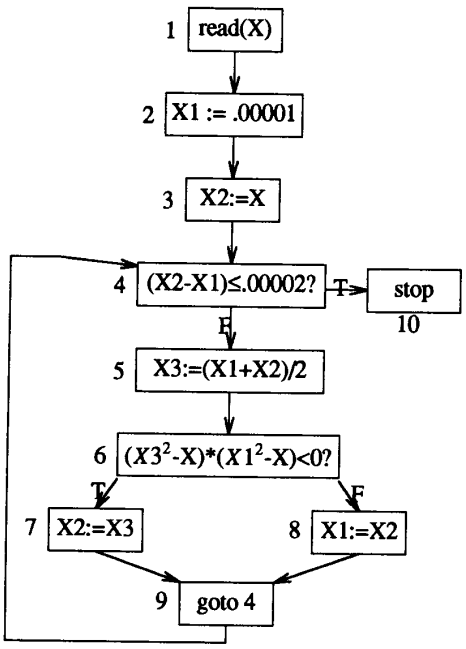
```
action DeleteEdge(e = p->t)
  V = set of program variables
  CFG is the original control flow graph
  CFG' is the modified control flow graph which excludes
       edge e = p->t
  DefsOfV = BackwardWalk(t,{V}) on CFG
  AfterTriples = ForwardWalk( {(s,def(s)): s ε DefsOfV}, false )
       on CFG'
  AfterPairs = {(def,use): (def,use,variable) ε AfterTriples}
  BeforeTriples = ForwardWalk( {(t,V_i): DefsOfV contains a
       definition of V_i and V_i ≠ def(t)}, true ) on CFG
  BeforePairs = {(s,t): s ε DefsOfV, def(s) ε ref(t)} ∪
       {s_i, s) : s_i ε DefsOfV and (p,s,V_i) ε BeforeTriples}
  DeletedPairs = BeforePairs - AfterPairs
  DefUsePairs = BeforePairs - DeletedPairs
end DeleteEdge
```

3. Testing Affected Def-Use Pairs

Next, we illustrate our regression testing technique through the example program whose control flow graph is shown in Figure 5. The program should compute the square root for input variable X by repeatedly halving the interval. The table on the right in Figure 5 gives the def-use pairs for the program that are required to satisfy the 'all-uses' data flow testing criterion. The program has an error in statement 8: statement 8 should read "X1:=X3". Correcting the error requires two actions: (1) delete the use of X2 in statement 8 and (2) insert the use of X3 in statement 8. We show how our algorithms handle the changes.

(1) Delete the use of X2 in statement 8.

ForwardWalk on X2 in statement 8 first finds the affected definition of X1 in statement 8. Then, uses of X1 in statements 4, 5 and 6 are detected. Thus, detected def-use pairs for X1 are (8,(4,6)), (8,(4,5)), (8,5), (8,(6,7)) and (8,(6,8)). Since the definition of X3 in statement 5 may be affected, def-use pairs for X3,



Def-use Information

node	*definition*	*def-use pairs*
1	X	(1,3), (1,6)
2	X1	(2,(4,5)), (2,(4,10)), (2,5), (2,(6,7)), (2,(6,8))
3	X2	(3,(4,5)), (3,(4,10)), (3,5)
5	X3	(5,(6,7)), (5,(6,8)), (5,7)
7	X2	(7,(4,5)), (7,(4,10)), (7,5)
8	X1	(8,(4,5)), (8,(4,10)), (8,5), (8,(6,7)), (8,6,8))

Figure 5. Program to compute the square root of X with an error in statement 8 that must be changed to "X1:=X3".

(5,(6,7)), (5,(6,8)) and (5,7) are detected. The use of X3 in statement 7 may affect the definition of X2 and def-use pairs for X2 (7,(4,5)), (7,(4,10)) and (7,5) are identified. Although the definition of X3 in statement 5 may be affected by this use of X2, the processing stops since def-use pairs for this definition of X3 are already detected.

(2) Insert the use of X3 in statement 8.

First, *BackwardWalk* on statement 8 for X3 finds the reaching definition of of X3 in statement 5 and returns the def-use pair (5,8) for X3. Since *ForwardWalk* on X3 in statement 8 is identical to the *ForwardWalk* performed during the above deletion of the use of X2 in statement 8, the same def-use pairs are returned. Table 1 lists the def-use pairs that are affected after the program change and must be tested.

node	*definition*	*def-use pairs*
5	X3	(5,(6,7)), (5,(6,8)) (5,7) (5,8)
7	X2	(7,(4,5)), (7,(4,10)), (7,5)
8	X1	(8,(4,6)), (8,(4,10)), (8,5) (8,(6,7)), (8,(6,8))

Table 1: Def-use Pairs for Regression Testing

After the affected def-use pairs are identified (see Table 1), test cases must be executed to exercise these def-use pairs. One technique is to maintain a test suite that is used to retest the program. If a test suite is maintained, test cases are selected from the the test suite to retest the def-use pairs after a program change and the test suite is updated to reflect the change. The test suite typically contains an association between each def-use pair and the test case used to test that pair. Information about the test case includes the input values and the execution path. The test suite is examined to determine the test cases that satisfy the affected def-use pairs and the changed code. These test cases are rerun and we check to see which affected pairs are satisfied by a test case, as well as updating the test suite. If a def-use pair does not have an associated test case, then a new test case can be generated. Our technique requires fewer test cases than previous methods since we do not execute all test cases that traverse the changed node.

To save the overhead involved in storing and updating test suites, another approach is to generate test cases only for the changed parts of the program during regression testing. Since our algorithms explicitly identify both directly and indirectly affected def-use pairs, we only need to run test cases that satisfy these pairs and we are still guaranteed to provide 'all-uses' data flow testing

coverage of the program. Thus, unlike other regression testing techniques, our method identifies affected def-use pairs without the overhead of maintaining and updating a test suite.

In either of the scenarios given above, new test cases may be required. Test cases can be generated manually by the user or automatically using a test case generator. When a new test case is required, a program slice can be computed based on the affected def-use pairs to assist in generation of test cases. This program slice contains those statements that are needed to traverse all affected and untested def-use pairs. If test cases are generated manually, the program slice enables the user to focus on the program statements pertinent to the affected def-use pairs. If the test cases are automatically generated, the test case generator only considers the program statement that are in the slice. In either case, using the slice reduces the test case generation effort since only the input variables on the slice are considered. Additional details of our slicing algorithm and its use in test case generation can be found in [4, 5].

4. Conclusion

We have presented a regression testing technique which utilizes slicing. The use of the slice for regression testing is efficient in terms of both memory and time overhead. Unlike previous regression techniques, we do not need to completely recompute data flow information nor maintain a history of previous data flow computation. We recompute the partial data flow that is needed, as driven by the program changes. Importantly, we do not need the expense of maintaining a test suite, including input, output and updates of the test suite. If the test suite is maintained, our approach reduces the number of test cases that must be rerun to provide full testing coverage and to update the test suite. Although we have presented our technique to satisfy the 'all-uses' criterion, it could easily be modified for other data flow testing criteria.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman, in *Compilers, Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, Massachusetts, 1986.
2. L. A. Clarke, A. Podgurski, D. Richardson, and S. Zeil, "A comparison of data flow path selection criteria," *Proceedings 8th International Conference on Software Engineering*, pp. 244-251, August 1985.
3. P. G. Frankl and E. J. Weyuker, "An applicable family of data flow testing criteria," *IEEE Transactions on Software Engineering*, vol. SE-14, no. 10, pp. 1483-1498, October 1988.
4. R. Gupta and M.L. Soffa, "A framework for generalized slicing," Technical Report, University of Pittsburgh, 1991.
5. R. Gupta and M.L. Soffa, "Automatic generation of a compact test suite," *Twelfth IFIP World Computer Congress*, September 1992.
6. M. J. Harrold and M. L. Soffa, "An incremental approach to unit testing during maintenance," *Proceedings of the Conference on Software Maintenance 1988*, pp. 362-367, October 1988.
7. M. J. Harrold, "An approach to incremental testing," Technical Report 89-1 Department of Computer Science, Ph.D. Thesis, University of Pittsburgh, January 1989.
8. B. Korel and J. Laski, "A tool for data flow oriented program testing," *ACM Softfair Proceedings*, pp. 35-37, December 1985.
9. S. C. Ntafos, "An evaluation of required element testing strategies," *Proceedings of the 7th International Conference on Software Engineering*, pp. 250-256, March 1984.
10. T. J. Ostrand and E. J. Weyuker, "Using data flow analysis for regression testing," *Sixth Annual Pacific Northwest Software Quality Conference*, pp. 58-71, September 1988.
11. A. M. Taha, S. M. Thebut, and S. S. Liu, "An approach to software fault localization and revalidation based on incremental data flow analysis," *Proceedings of COMPSAC 89*, pp. 527-534, September 1989.
12. M. Weiser, "Program slicing," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 352-357, July 1984.
13. E. J. Weyuker and T. J. Ostrand, "Theories of program testing and the application of revealing subdomains," *IEEE Transactions on Software Engineering*, pp. 236-246, May 1980.
14. E. J. Weyuker, "The cost of data flow testing: An empirical study," *IEEE Transactions on Software Engineering*, vol. SE-16, no. 2, pp. 121-128, February 1990.