Program Slicing-Based Regression Testing Techniques†

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

Mary Jean Harrold
Dept. of Computer and Information Science
Ohio State University
Columbus, OH 43210-1277
harrold@cis.ohio-state.edu

Mary Lou Soffa
Dept. 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 on the modifications 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 algorithms to explicitly detect definition-use associations that are affected by a program change. An important benefit of our slicing technique is, unlike previous techniques, neither data flow history nor recomputation of data flow for the entire program is required to detect affected definition-use associations. 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 with respect to the affected definition-use associations 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 have been completely tested during its development to satisfy some adequacy criterion, program changes during maintenance require that parts of the software be retested. *Regression testing* is the process of validating 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. There are two important problems in selective regression testing: (1) identifying those existing tests that must be rerun since they may exhibit different behavior in the changed program and (2) identifying those program components that must be retested to satisfy some coverage criterion. This work focuses on the second problem in that it identifies program components to satisfy data flow testing criteria for the changed program.

Techniques for selective regression testing that use the data flow in a program to identify program components to retest after changes[9, 10, 14, 19] have been developed. In data flow testing [3, 6, 12, 13], a variable assignment is tested (*satisfied*) by tests 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) associations, and test data adequacy criteria are used to select particular def-use associations to test. Tests are then generated that cause execution through these selected def-use associations. Selective regression testing techniques for data flow testing first identify the def-use associations that are affected by a change and then select tests that satisfy these affected def-use associations.

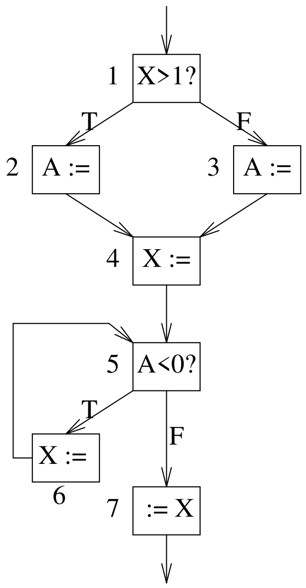
† Partially supported by the NSF through Presidential Young Investigator Award CCR-9157371 and Grant CCR-9402226 to the University of Pittsburgh, and Grant CCR-9109531 and National Young Investigator Award CCR-9357811 to Clemson University.
An earlier version of this paper appeared in *Proceedings of the Conference on Software Maintenance*, November 1992.

Existing data flow based regression testing techniques explicitly identify *directly* affected def-use associations by detecting *new* def-use associations that are created by a program change. These techniques compute the changed data flow by either (1) incrementally updating the original data flow to agree with the modified code [9, 10, 19] or (2) exhaustively computing the data flow for both the original and modified programs and comparing the sets to determine the differences. Thus, these techniques either save data flow information between testing sessions or completely recompute it at the beginning of each session. A program change can also *indirectly* affect def-use associations due to either a change in the computed value of a definition or a change in the predicate value of a conditional statement. Existing regression testing techniques identify these indirectly affected def-use associations by either running all tests from the test suite that previously executed through the changed code[9, 10, 19], or using slicing techniques that require prior computation of the data flow information for the program[2, 18].

This paper presents a new approach to selective regression testing using the concept of a *program slice*. A backward program slice [11, 20] at a program point p for variable v consists of all statements in the program, including conditionals, that might affect the value of v at p, whereas a forward program slice [11] at a program point p for variable v consists of all statements in the program, including conditionals, that might be affected by the value of v at p. The technique uses two slicing algorithms to determine directly and indirectly affected def-use associations. 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 uses, and subsequent definitions and uses, that are affected by a definition that is changed as a result of the program edit. Additionally, the algorithm identifies any def-use associations that depend on a changed predicate. Through these two algorithms, this technique detects def-use associations that are changed or affected because of program modifications.

The slicing algorithms are efficient in that they detect the def-use associations without requiring either the data flow history or the complete recomputation of data flow for the entire program. These algorithms are based on the approach taken by Weiser[20] that uses the control flow graph representation of the program and only requires the computation of partial data flow information. Unlike previous regression testing techniques that require either a test suite or data flow information to select tests for regression testing, this technique explicitly identifies all affected def-use associations. Thus, the technique requires neither a test suite nor complete data flow information to enable selective retesting. If a test suite is maintained, fewer tests may be executed since only those tests that may execute affected def-use associations are rerun.

The next section presents background for the slice-based technique. Section 3 describes the analysis required to detect affected def-use associations and presents the algorithms for backward and forward walks. Section 3 also presents the algorithm that uses the backward and forward walks to determine what to retest for different types of program edits, along with our general approach for translating higher level edits to lower level edits. Section 4 discusses the merit of the technique and its applications to testing. Concluding remarks are given in Section 5.



Definition-c-use Associations

(4, 7, X)
(6, 7, X)

Definition-p-use Associations

(2, (5,6), A)
(2, (5,7), A)
(3, (5,6), A)
(3, (5,7), A)

Figure 1. Definition-c-uses for variable X and definition-p-uses for variable A.

2. Background

This section overviews data flow testing, the basis of the regression testing technique. The technique also uses control dependence information to identify which affected def-use associations in a program to retest. Thus, this section also briefly discusses control dependence.

2.1. Data Flow Testing

Several data flow testing techniques [6, 12, 13] 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], based on a control flow graph representation of a program, are used to compute def-use associations. In a control flow graph, each node 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 uses these definitions and uses to compute def-use associations. 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. Def-use associations are represented by triples, (s, u, v), where the value of variable v defined in statement s is used in statement or edge u. In Figure 1, definitions are shown with the variable on the left side of the assignment and uses are shown with the variable on the right side of an assignment or in a predicate. In the figure, node 7 contains a c-use of the definition of X in node 4. The triple (4, 7, X) represents this def-use association. Node 7 also contains a c-use of the definition of X in node 6, and the triple (6, 7, X) represents this def-use association. Figure 1 contains a p-use in node 5 of the definitions of A in nodes 2 and 3, so there are def-use associations between the definitions in nodes 2 and 3 and each of the edges leaving conditional node 5; triples (2, (5,6), A), (2, (5,7), A) (3, (5,6), A) and (3, (5,7), A) represent these def-use associations.

Test data adequacy criteria are used to select particular def-use associations to test. One criterion, all-du-paths, requires that each definition of a variable be tested on each loop-free subpath to each reachable use. Another criterion, all-uses, requires that each definition of a variable be tested on some subpath to each of its uses. Other criteria, such as all-defs, require that fewer def-use associations be tested. For a complete discussion of data flow testing, see references [6, 16].

2.2. Control Dependence

To identify def-use associations that may be affected by a program change, the slice-based approach uses *control dependence* information. Informally, a node (statement) in a control flow graph is control dependent on another node (statement) Y if there are two paths out of Y, such that one path necessarily reaches X and the other path may not reach X. This definition of control dependence, given by Ferrante, Ottenstein and Warren [5], is essentially the same as the definition of *direct strong control dependence* given by Clarke and Podgurski [15].

In Figure 1, the execution of statement 2 depends on statement 1 evaluating to *true*, whereas the execution of statement 3 depends on statement 1 evaluating to *false*. Thus, statements 2 and 3 are control dependent on statement 1. Also, statements 6 and 7 are control dependent on statement 5. However, statements 1, 4, and 5 are only control dependent on reaching the point immediately before statement 1.

3. Detecting Affected Definition-Use Associations

To satisfy a data flow testing criterion after making a program change requires identifying the def-use associations affected by the change. This section first discusses the different types of affected def-use associations. Then, it discusses the slicing algorithms for forward and backward walks that identify the definitions and uses that are affected by a program edit. Next, the section presents the algorithm to handle the different types of program edits, and finally, it discusses the way in which higher level edits are translated to lower level edits for processing.

3.1. Types of Affected Def-Use Associations

Affected def-use associations fall into two categories: (1) those affected *directly* because of the insertion/deletion of definitions and uses (*new associations*) and (2) those affected *indirectly* because of a change in either a computed value (*value associations*) or a path condition (*path associations*).

<u>*New Associations*</u>: A program edit creates new def-use associations that must be tested. For example, consider the following code segment:

```
   . . .
1. if A > 1 then
2.     Y := X + 5
3. else
4.     Y := X - 5
5. endif
6. X := 2                    /* replace with “X := 2 + Y” */
   . . .
```

If statement 6 is replaced with “X:=2+Y”, a new use of variable Y is introduced. Def-use associations consisting of those definitions of Y that reach the new use of Y must be tested. These new associations are (2, 6, Y) and (4, 6, Y).

Value Associations Value associations are def-use associations whose computed values may have changed because of the program edit and therefore, require retesting. For example, consider the following code segment:

```
    . . .
1.  X := 2              /* replace with "X := 3" */
2.  if A > 1 then
3.      Y := X + 5
4.  else
5.      Z := X - 5
6.  endif
7.  T := Y + 6
8   U := Z + B
    . . .
```

If statement 1 is replaced with “X := 3”, no new def-use associations are created. However, the def-use associations that depend on the new value of X are retested since they are affected by the change. Since both statements 3 and 5 use the definition of X in statement 1, def-use associations (1, 3, X) and (1, 5, X) are value associations. The new value of X in statement 3 affects the computed value of Y at that statement, which causes value association (3, 7, Y) to be identified. Likewise, value association (5, 8, Z) is found because of the affected value of Z in statement 5. Now, the values of T in statement 7 and U in statement 8 are affected, and the process of identifying value associations continues with uses of these variables.

Path Associations The def-use associations that are affected on a path whose path condition has changed must be retested. A path condition can be altered because of an explicit change in an operator in the predicate statement. For example, consider the following code segment:

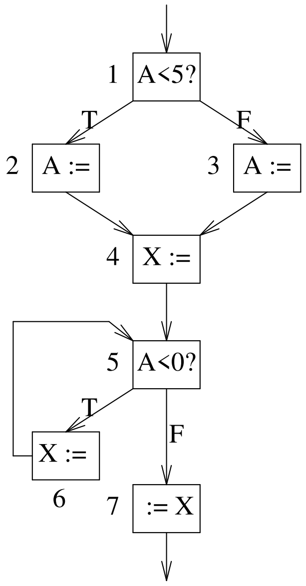
```
    . . .
1.  X := 2              /* replace with "X := 3" */
2.  if A > 1 then       /* replace with "A < 1" */
3.      Y := X + 5
5.  endif
6.  if X > A then
7.      Y := X - 5
8.  endif
9.  write Y
    . . .
```

If statement 2 is changed to “if A < 1 then”, then no new def-use associations are created, but any def-use association that is control dependent on statement 2 may be affected by the change. Here, statement 3 is control dependent on statement 2, and so path association (3, 9, Y) is identified.

A path condition can also be altered because of a change in the value of a p-use in a predicate. For example, in the above code segment, if statement 1 is replaced with “X := 3”, then the value computed in statement 6 is affected, and the path condition to statement 7 is changed. Thus, path association (7, 9, Y) is found.

3.2. 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 a control flow graph representation of the program in which each node represents a single statement. These algorithms compute data flow information to identify affected def-use associations but require no past history of data flow information. Furthermore, the algorithms are slicing algorithms in that they examine only



BackwardWalk(7,{X})

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

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

Figure 2. *BackwardWalk* on variable X at statement 7. *BackwardWalk* locates definitions of X in nodes 4 and 6 that reach statement 7.

relevant parts of the control flow graph to compute the required data flow information. These algorithms are designed based on the approach taken by Weiser for computing slices[20]. This approach lets relevant program slices be computed without exhaustively computing the def-use information for the program. This discussion assumes 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.

3.3. Algorithm *BackwardWalk*

The backward walk algorithm identifies statements containing definitions of variables that reach a program point. Before the algorithm is presented in detail, it is demonstrated with an example. Figure 2 gives a program segment containing some definitions and uses. Suppose the goal is to compute the set of statements containing definitions of variable X that reach statement 7. The walk begins at the point immediately before statement 7 in the control flow graph. A walk back over edges (5,7) and (4,5) locates statement 4, which contains a definition of X. Since no other definition of X can reach statement 7 along a path through statement 4, the search for definitions of X stops at statement 4. Since there are two backward paths from statement 5, the algorithm also walks back over edge (6,5) and finds statement 6 that contains a definition of X. The backward walk algorithm locates the statements containing definitions of variable X that reach a point in the program, without computing data flow analysis for the entire program. The technique uses the definitions that reach the statement, along with the uses in the statement, to form def-use associations. It should be noted that if a variable being considered is undefined along a path, then the search will terminate once it reaches the start node of the control flow graph.

```
algorithm BackwardWalk(s,U)
input       s : program point/statement
            U : set of program variables
output      DefsOfU : set of statements/nodes in the control flow graph
declare     In[i], Out[i], NewOut: set of program variables
            Worklist : statements/nodes in the control flow graph, maintained as a priority queue
            n, n_i : program point/statement
            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 +_rdf Worklist     /* rdf = reverse-depth-first */
    In[s] = U;   Out[s] = ∅
    forall n_i ≠ s  do In[n_i] = Out[n_i] = ∅

    while Worklist ≠ ∅ do                                 /* more statements along backward walk, continue processing */
        Get n from head of Worklist                       /* get next statement in backward walk */
        NewOut =   ∪        In[p]                         /* recompute Out set as union of In sets of successors */
                pεSucc(n)
        if NewOut ≠ Out[n] then                           /* there is change from last iteration */
            Out[n] = NewOut                               /* assign new out set to Out[n] */
            if n defines a variable u ε U then            /* a definition of variable in U is found along this path */
                DefsOfU = DefsOfU ∪ {n}                   /* add statement n to definitions set */
                In[n] = Out[n] - {u}                      /* stop searching for definitions of u */
            else  In[n] = Out[n]                          /* there is no definition in n , just propagate */
            if In[n] ≠ ∅ then                             /* all definitions of variables in U not found, */
                                                          /* add Preds(n) to Worklist */
                forall x ε Preds(n) do Worklist = x +_rdf Worklist
    return(DefsOfU)                                       /* all statements containing definitions of variables in U */
end BackwardWalk
```

Figure 3. Algorithm *BackwardWalk* computes the definitions of variables in *U* that reach the statement *s*.

Algorithm *BackwardWalk*, given in Figure 3, identifies the statements containing definitions of a set *U* of variables that reach a program point *s*. *BackwardWalk* inputs the program point or statement *s* and a set *U* of program variables, and outputs *DefsOfU*, a set of statements or nodes in the control flow graph corresponding to the definitions of variables in *U* that reach *s*. *BackwardWalk* traverses the control flow graph in the backward direction from *s* all variables of *U* are encountered along each path. The algorithm collects the statements containing the definitions in *DefsOfU* and returns the set.

To assist in the traversal process, *BackwardWalk} maintains sets of variables,In* and *Out*, for relevant nodes in the control flow graph. *Out*[*i*] contains the variables whose definitions the algorithm has not encountered along some path from the point immediately following *i* to *s*; *In*[*i*] contains the variables whose definitions the algorithm has not encountered along some path from the point immediately preceding *i* to *s*. Since the algorithm walks backward in the control flow graph, it computes *Out*[*n*] as the union of the *In* sets of *n*'s successors. The algorithm uses another set of variables, *NewOut*, to store temporarily the newly computed *Out* set. During the backward traversal, the algorithm maintains a worklist, *Worklist*, consisting of those nodes that must be visited; *Worklist* indicates how far the traversal has progressed. *BackwardWalk maintains Worklist* as a priority queue based on a reverse depth first ordering of nodes in the control flow graph. The algorithm also uses n and n_i to represent statements or nodes in the control flow graph, and functions *Pred(i)* and *Succ(i)* to compute the immediate predecessors and successors of node *i*, respectively.

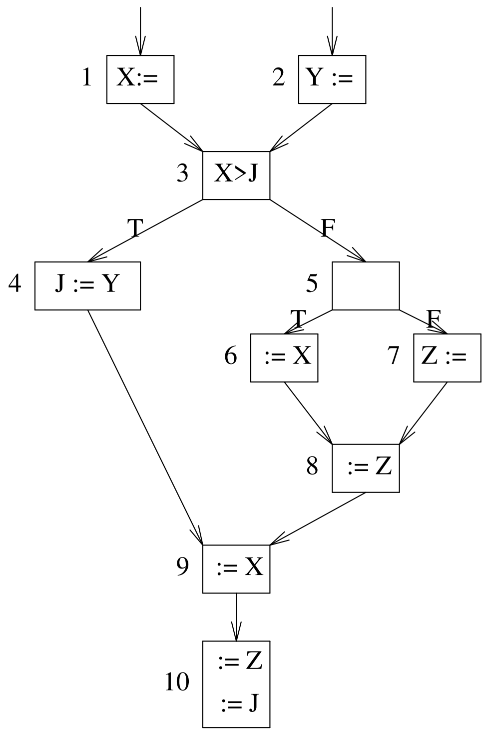
Algorithm *BackwardWalk* begins by initializing all sets that it uses. After initialization, the only entries in *Worklist* are the predecessors of *s*. The main part of the algorithm is a **while** loop that repeatedly processes statements in *Worklist* until *Worklist* is empty. To process a statement *n*, *BackwardWalk* first computes *NewOut* for *n* as the union of the *In* sets of the successors of *n* in the control flow graph. If *NewOut* and *Out[n]* are the same, there has been no change from the last iteration of the **while** loop, and processing along the path containing *n* terminates; the comparison of *NewOut* and *Out[n]* causes each loop to be processed only one time. If *NewOut* and *Out[n]* differ, there is a change from the last iteration of the **while** loop. In this case, *BackwardWalk* assigns *NewOut* to *Out[n]*, and examines *n* for a definition of a variable in *U*. If the algorithm finds such a definition, it adds *n* to *DefsOfU*. Additionally, the algorithm removes *u* from *In[n]* since it no longer needs to search for a definition of *u* along this path. If *BackwardWalk* finds no definition of a variable in *U* in *n*, it assigns *Out[n]* to *In[n]* and adds all immediate predecessors of *n* to *Worklist*. Each statement *n* added to *Worklist* represents a point in the program along which the backward traversal must continue, since not all variables in *U* were defined along a path from a successor of *n* to point *s*. Thus, *BackwardWalk* only adds a node to *Worklist* if the *In* set of one of its successor is not empty. When *Worklist* is empty, the algorithm has encountered all definitions of all variables in *U* along all backward paths from *s* and the algorithm terminates.

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

In the following analysis of the runtime complexity of the *BackwardWalk* algorithm, let *s* represent the number of nodes in the control flow graph. The identification of each member of *DefsOfU* may in the worst case require the traversal of the entire control flow graph. There are two components involved in the processing of each node encountered during the traversal: (1) inserting the node in *Worklist*, which takes O(log s) time since *Worklist* is a priority queue and (2) processing the node when it reaches the front of the *Worklist*, which takes constant time assuming that bit-vectors are used, where each bit represents a distinct variable in the program. Thus, the overall runtime complexity of *BackwardWalk* is bounded by O(s log s |*DefsOfU*|).

3.3.1. Algorithm *ForwardWalk*

The forward walk algorithm identifies uses of variables that are directly or indirectly affected by either a change in a value of a variable at a point in the program or a change in a predicate. The def-use associations returned by the algorithm are triples (*s, u, v*) indicating that the value of variable *v* at statement *s*, affected by the change, is used by statement *u*. A def-use association is directly affected if the triple represents a use of an altered definition. A def-use association is indirectly affected in one of two ways: (1) the triple is in the transitive closure of the changed definition, or (2) the triple is control dependent on a changed or affected predicate.



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

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

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

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

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

Consider the program segment in Figure 4. If a forward walk begins at statement 2 for variable Y, the use of Y in statement 4 is found. Thus, directly affected def-use association (2,4,Y) is computed. Additionally, since def-use association (4,10,J) for variable J is in the transitive closure of the definition of Y in statement 2, this def-use association is indirectly affected. If a forward walk begins at statement 1 for variable X, the uses of X in statements 3, 6 and 9 are located, and directly affected def-use associations (1,3,X), (1,6,X) and (1,9,X) are computed. Additionally, because of the affected predicate in statement 3, any def-use association whose definition is control dependent on statement 3 is indirectly affected. Thus, def-use associations (7,8,Z) and (7,10,Z) for variable Z are identified as indirectly affected.

Algorithm *ForwardWalk*, given in Figure 5, inputs a set of *Pairs* representing definitions whose uses are to be found, along with a boolean, *Names*, that indicates whether the walk starts with a set of variable names at a program point† or a set of definitions. *Names* is true if the walk begins with a set of variable names v at a point p, represented by (p, v). Otherwise, the walk begins with the pairs of affected definitions (s, v). *ForwardWalk* outputs a set of def-use triples, *Triples*.

† *ForwardWalk* can handle multiple pairs, consisting of points/statements and variables of the form (*s*,*v*), by simultaneously processing all of these pairs. To simplify the discussion, *ForwardWalk* is described for a single pair (*s*,*v*).

```
algorithm ForwardWalk(Pairs, Names)
input     Names : boolean is true if change is only a predicate change
          Pairs: sets of definitions, (s, v), where s is a program point/statement and v is a variable
output    Triples : set of (point/statement, statement, variable)
declare   In[i], Out[i], Kill, NewIn : set of pairs, (point/statement,variable)
          Worklist, Cd[i], PredCd, AffectedPreds : set of point/statement
          DefsOfV : set of (s, v) of definitions
          v : program variable
          k, n : statement/node
          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
    Triples = ∅                                                      /* initialization */
    forall (s,v) ε Pairs do
        forall n ε Succ(s) do Worklist = n +_df Worklist             /* df = depth first */
    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 Names then AffectedPreds = {s_i} else AffectedPreds = ∅

    while Worklist ≠ ∅ do                                            /* continue processing nodes*/
        Get n from head of Worklist
        NewIn = ∪_{pεPred(n)} Out[p]
        if NewIn ≠ In[n] then                                        /* change from last iteration */
            In[n] = NewIn                                            /* recompute In[n] */
            PredCd = ∪_{pεPred(n)} Cd(p)
            if PredCd - Cd(n) ≠ ∅ then UpdateAffInfo                 /* update affected predicate information */
            if n has a c-use of variable v such that (d,v) ε In[n] then   /* found a c-use */
                forall (d,v) ε In[n] do Triples = Triples ∪ {(d,n,v)}     /* additional def-use associations */
                Kill = {(s,Def(n)): (s,Def(n)) ε In[n]}              /* definitions that are killed by n */ */
                Out[n] = (In[n] - Kill) ∪ {(n,Def(n))}               /* propagate definitions to end of n */
            elsif n has a p-use of variable v such that (d,v) ε In[n] then   /* found a p-use */
                forall (d,v) ε In[n] do Triples = Triples ∪ {(d,n,p)}     /* additional def-use associations */
                DefsOfV = BackwardWalk(n, {v}) - In[n]               /* find definitions that reach affected predicate */
                In[n] = In[n] ∪ {(n, v_i) : (d, v_i) ε DefsOfV}      /* add new definitions to In and Out sets */
                Out[n] = In[n]
                AffectedPreds = AffectedPreds ∪ {n}                  /* mark this predicate statement as affected */
            elseif n defines a variable v Λ Cd(n) ∩ AffectedPreds ≠ ∅ then   /* statement contains a definition */
                Out[n] = Out[n] ∪ {(n, Def(n))}                      /* propagate this definition forward */
            else Out[n] = In[n]                                      /* statement contains no definition or use of interest */
            if Out[n] ≠ ∅ then                                       /* more definitions/uses, continue processing */
                forall x ε Succ(n) do Worklist = x +_df Worklist
    return(Triples)                                                  /* all affected value and path associations */
end ForwardWalk
```

Figure 5. Algorithm *ForwardWalk* identifies all def-use triples that are affected by a change in the value of variable v at point s in the program, or that are affected by a predicate change.

For each statement node n, *In* and *Out* sets contain the pairs representing 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 *ForwardWalk* encounters a statement n that uses the value (d,p) belonging to $In[n]$, it adds def-use triple (d,n,p) 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 defined by statement n is also indirectly affected. If the algorithm encounters a new definition of a variable p 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's predecessors. During this traversal, a worklist, *Worklist*, consisting of those nodes that must be visited, indicates how far the traversal has progressed. *ForwardWalk* maintains *Worklist* as a priority queue based on a depth first ordering of nodes in the control flow graph. The algorithm examines the statements in *Worklist* for c-uses and p-uses of the definitions in the *In* sets along with definitions in statements that are control dependent on a changed or affected predicate. As the algorithm examines the statements in *Worklist*, it adds additional statements to be considered to *Worklist*. *ForwardWalk* also uses *DefsOfV*, a set of definitions, v, a program variable, and k and n, statements in the program. Functions *Pred*, *Succ* and *Def* return the predecessors, successors and variable defined by statement i, respectively.

In the first part of *ForwardWalk*, all variables are initialized. The main part of the algorithm is a **while** loop that processes statements/nodes on *Worklist* until *Worklist* is empty. For each statement n removed from *Worklist*, processing consists of first computing *NewIn* for n by taking the union of the *Out* sets of the predecessors of n, and then determining if *NewIn* is the same as the previous value of *In[n]*. If these sets are the same, there has been no change since the previous iteration, and the forward walk along this path terminates at n. If these sets differ, n is processed further: *NewIn* is assigned to *In[n]*, and *PredCd* is assigned the union of the control dependence information for n's predecessor(s). If *PredCd* contains nodes on which n is not control dependent, then the forward walk along this path has moved into a different region of control dependence, and *AffectedPreds* must be updated accordingly. Procedure *UpdateAffInfo* shown in Figure 6 (described below) handles this task. Then, node n is checked to see if it has a c-use of variable v. If so, def-use associations for any pairs in *In[n]* are added to *Triples*, and the variable defined at n is added to *Out[n]* since it is indirectly affected. If there is no c-use of v at n but n contains a p-use, def-use associations for any pairs in *In[n]* are added to *Triples*. Since a p-use signals an affected predicate, any definitions that reach n are found using *BackwardWalk*; these definitions are used to identify indirectly affected def-use associations. If neither type of use is found at n, the statement is inspected for a definition; if one is found at n, the appropriate data flow sets are updated. Finally, if no definition or use of the variables is found at n, the data flow information is propagated through n. If *Out[n]* is not empty, then the successors of n must be processed, and they are added to *Worklist*. When *Worklist* is empty, processing terminates and *Triples* is returned.

An important feature of *ForwardWalk* is that it identifies def-use associations that are control dependent on an affected predicate, even though the value computed by the definition is unaffected. Thus, the control dependence information must be computed prior to using *ForwardWalk*. Control dependencies are efficiently computed for each node in a control flow graph using the post-dominator relation among the nodes [5]. 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

```
procedure UpdateAffInfo
begin
    forall k ε (PredCd - Cd(n)) ∩ AffectedPreds do      /* statement is out of previous control dependence region */
        In[n] = In[n] - {(k, v_i) for all v_i}          /* update In[n] accordingly */
        AffectedPreds = AffectedPreds - {k}             /* k is no longer affected in new region */
        forall (k, u, v) in Triples do                  /* update def-use associations */
            Triples = Triples - {(k, u, v)}
            forall (d, v) ε DefsOfV do Triples = Triples ∪ {(d, u, v)}
end UpdateAffInfo
```

Figure 6. Procedure *UpdateAffInfo* adjusts predicate information when the walk has entered a different region of control dependence.

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 associations that are control dependent on affected predicates.

ForwardWalk must also recognize all def-use associations that are control dependent on affected predicates. To accomplish this, each time a new node is removed from *Worklist*, its control dependencies are compared with the control dependencies of its predecessors. If there is a difference in control dependence information, the procedure *UpdateAffInfo*, shown in Figure 6, is called. Assume that *UpdateAffInfo* can access all variables in *ForwardWalk*. *UpdateAffInfo* considers those affected predicates on which n is not control dependent, and removes these statements from *In[n]*. A decrease in control dependencies indicates that some predicate may no longer be affected; thus it is removed from *AffectedPreds* and *Triples* is 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 yield the def-use association (2,4,Y). Since definition J in statement 4 is affected by the change in Y, *ForwardWalk* adds (4,J) to *OUT[4]* and continues the traversal to find the def-use association (4,10,J). In the second *ForwardWalk*, the walk 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 *Triple* (3,6,X) that is used to identify affected def-use association (1,6,X). However, def-use association (1,9,X) 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,J) is identified. Since statement 7 is control dependent on statement 3 and contains a definition, def-use associations (7,8,Z) and (7,10,Z) are identified for Z even though their values are not affected by the change in statement 3.

In the following analysis of the runtime complexity of the *ForwardWalk* algorithm, let s represent the number of nodes in the control flow graph. The identification of each member of *Triples* is achieved in one of the following two ways: (1) a forward traversal of the flow graph to identify a use of a value, or (2) a backward traversal from a predicate node, followed by a forward traversal from the same predicate node. In the worst case, the forward/backward traversals may inspect the entire control flow graph and thus process all nodes. The processing of each node consists of two components: (1) inserting the node in *Worklist* which takes $O(logs)$ time and (2) processing the node when it reaches the front of the *Worklist*, which takes $O(s)$ time since it requires examining each element in the node's data flow set, and the size of the data flow set is bounded by n. Thus, the overall runtime complexity of the *ForwardWalk* algorithm is $O(s^2|Triples|)$.

algorithm *FindDUAssns*(*S*, *Edit*)
input *S* : program point/statement
Edit: Insert (Delete) use(definition), Change statement (branch) operator, Insert(Delete) edge
output *DefUseAssns* : set of (point/statement, statement, variable)
declare *DefsOfX* : set of statements/nodes in the control flow graph
NewAssns, ValuePathAssns, Triples, Triples' : set of (point/statement, statement, variable)
CFG, CFG' : control flow graph
V : set of program variables
DefsOfV, DefsOfY, DefsOfVp, DefsOfVt : set of variable definitions
Def(i) : returns the variable defined by statement *i*

begin
case *Edit* **is** :
Insert a use of variable *X* in statement $S: Y := ..X..$: *DefUseAssns* := *InsertUse(S,X)*
Delete a use of variable *X* in statement $S: Y := ..X..$: *DefUseAssns* := *DeleteUse(S,X)*
Insert a definition of variable *Y* in statement $S: Y := ..$: *DefUseAssns* := *InsertDefinition(S,Y)*
Delete a definition of variable *Y* in statement $S: Y := ..$: *DefUseAssns* := *DeleteDefinition(S,Y)*
Change operator in statement $S: Y :=$: *DefUseAssns* := *OperatorChange(S,Y)*
Change operator in branch condition in statement *S* : *DefUseAssns* := *ConditionChange(S)*
Insert an edge from statement p to statement t : *DefUseAssns* := *InsertEdge(p,t)*
Delete an edge from statement p to statement t : *DefUseAssns* := *DeleteEdge(p,t)*
end *FindDUAssns*

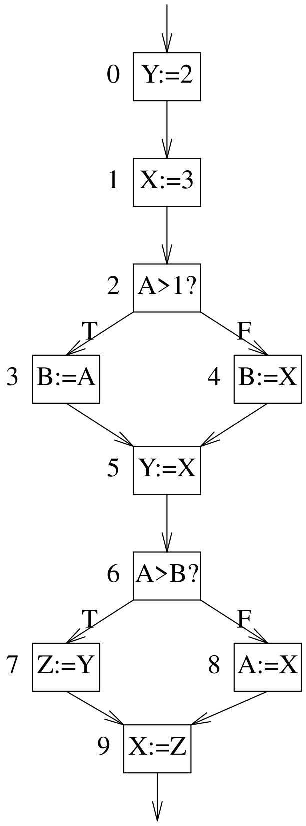
Figure 7. Algorithm *FindDUAssns* accepts a statement *S* and an edit type *Edit*, and identifies all def-use associations that must be tested because of that edit.

3.4. Actions for Different Types of Edits

This section considers consider the way in which we use *BackwardWalk* and *ForwardWalk* to identify the def-use pairs that are affected by a program change. Algorithm *FindDUAssns(S, Edit)*, given in Figure 7, inputs statement *S* where the edit occurs, along with the type of edit *Edit* that occurred at *S*. After processing the edit, *FindDUAssns* returns the def-use associations, *DefUseAssns*, that are affected by the edit. *FindDUAssns* uses several variables during its processing. *DefsOfX* stores statements/nodes in the control flow graph representing definitions in the program. *NewAssns*, *ValuePathAssns*, *Triples* and *Triples′* represent sets of def-use associations. *CFG* and *CFG′* represent control flow graphs, *V* represents a set of program variables, and *DefsOfV*, *DefsOfY*, *DefsOfVp* and *DefsOfVt* represent sets of variable definitions. Finally, $Def(i)$ is a function that returns the variable defined by statement i. Each action has access to this set of global variables. In the following discussion, we detail the action (function) taken as a result of each edit.

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

Inserting a use of variable *X* causes new def-use associations between statements containing definitions of *X* that reach the new use at *S*. First, function *InsertUse* uses *BackwardWalk* to locate all definitions of *X* that reach the new use of *X* in *S*; these definitions are returned in *DefsOfX*. Each of the definitions in *DefsOfX* together with the use of *X* in *S* represents a newly created def-use association that is added to *NewAssns*. Then, *ForwardWalk* computes *ValuePathAssns*, which are def-use associations that have experienced value changes because of the change in the value of *Y* at *S*, or def-use associations that are control dependent on an affected predicate. Finally, the set of affected def-use associations, *DefUseAssns*, is computed as the union of *NewAssns* and *ValuePathAssns*.



Change1: insert use of X in statement 5

BackwardWalk(5,{X}): definition of X in statement 1 reaches statement 5; def-use association to be tested is (1,5,X)
ForwardWalk({(5,Y)}, false): def-use associations to be retested are (5,7,Y) and (7,9,Z)

Change2: delete use of X in statement 5

ForwardWalk({(5,Y)}, false): def-use associations to be retested are (5,7,Y) and (7,9,Z)

Change3: insert definition of Y in statement 5

ForwardWalk({(5,Y)}, false): def-use associations to be retested are (5,7,Y) and (7,9,Z)

Change4: delete definition of Y in statement 5

BackwardWalk(5,{Y}): definition of Y in statement 0 reaches statement 5
ForwardWalk({(5,Y)}, false): def-use associations to be retested are (0,7,Y) and (7,9,Z)

Figure 8. Change1 inserts a use of X in statement 5. *BackwardWalk* identifies the def-use associations to the new use of X and *ForwardWalk* identifies the def-use associations that are affected by the change in the definition of Y in statement 5. Change2 deletes the use of X in statement 5. *ForwardWalk* identifies the def-use associations affected by this change in the definition of Y. Change3 inserts a definition of Y in statement 5. *ForwardWalk* identifies the def-use associations to be retested. Change4 deletes the definition of Y in statement 5. *BackwardWalk* identifies the definitions of X that reach statement 5 to form new def-use associations to be tested. Then, *ForwardWalk* identifies the def-use associations affected by the change in statement 5.

---

function *InsertUse(S,X)* /* use the modified CFG */
  DefsOfX = BackwardWalk(S,{X}) /* find statements with definitions of *X* that reach *S* */
  NewAssns = $\bigcup_{S_i \varepsilon DefsOfX} \{(S_i, \mathrm{S}, \mathrm{X})\}$ /* form new def-use associations */
  ValuePathAssns = ForwardWalk({S, Y)}, false) /* find indirectly affected def-use associations */
  DefUseAssns = NewAssns $\bigcup$ *ValuePathAssns* /* combine new and affected def-use associations */
end *InsertUse*

To illustrate *InsertUse*, consider Change1 in Figure 8; assume that the use of *X* in statement 5 has just been inserted. *BackwardWalk* on (5,{X}) finds the definition of *X* in statement 1, and (1,5,X) is added to *NewAssns*. Then, *ForwardWalk* on (5,Y) identifies the *ValuePathAssns* (5,7,Y) and (7,9,Z) because the change in the value of *Y* affects these def-use associations.

Delete a use of variable X in statement S: $Y := \ldots X \ldots$

Deleting a use of X causes the value of Y to change but introduces no new def-use associations. Function *DeleteUse* uses *ForwardWalk* to identify all def-use associations that are affected by the change in the value of Y. If these def-use associations involve predicate statements, then the affected path associations for those predicates are also identified during *ForwardWalk*. The returned set of def-use associations is assigned to *DefUseAssns*.

```
function DeleteUse(S, X)                          /* use the modified CFG */
   DefUseAssns = ForwardWalk({(S,Y)}, false)      /* find all affected def-use associations */
end DeleteUse
```

For example, if the use of X in statement 5 in Figure 8 (Change2) is deleted, *DeleteUse* uses *ForwardWalk* on (5,Y) to detect the def-use associations (5,7,Y) and (7,9,Z) that experience value changes; no affected path associations are found.

Insert a definition of variable Y in statement S: $Y := \ldots$

When a definition of Y is inserted in statement S, there are new def-use associations between S and statements containing reachable uses of Y; there are also affected def-use associations due to the change in the computed value at S. *InsertDefinition* uses *ForwardWalk* to identify the newly created def-use associations for the inserted definition of Y. Other definitions that experience value changes because of the change in S are also found. If any of the def-use associations involve predicates, the def-use associations affected by those predicates are also identified during *ForwardWalk*. The set of all new and affected def-use associations returned by *ForwardWalk* is assigned to *DefUseAssns*.

```
function InsertDefinition(S, Y)                   /* use the modified CFG */
   DefUseAssns = ForwardWalk((S, Y), false)       /* find all affected def-use associations */
end InsertDefinition
```

Assume that statement 5 in the example of Figure 8 is an inserted definition of Y (Change3). *ForwardWalk* yields new association (5,7,Y) because the newly computed value of Y is used in statement 7. Additionally, *ForwardWalk* identifies value association (7,9,Z) because the value of Z in statement 7 is affected.

Delete a definition of variable Y in statement S: $Y := \ldots$

When a definition of variable Y is deleted from statement S, there are new def-use associations since other definitions of Y now reach uses that were previously blocked by the deleted definition of Y. Additionally, there are def-use associations affected by the change in the computed value at S. *DeleteDefinition* uses *BackwardWalk* to identify the definitions of Y that reach S, the statement containing the deleted definition of Y, and adds them to *DefsOfY*. Now that the definition of Y in S is deleted, the definitions of Y in *DefsOfY* reach uses previously reached by the definition of Y in S. Additionally, the value and path associations that are affected by the deleted definition of Y are computed. Then, *ForwardWalk* identifies the def-use associations that are created by the edit at statement S, the value associations whose values are affected, and path associations dependent on affected predicates; these associations are returned in *Triples*. Those uses of Y reachable from S are associated with the definitions of Y in *DefsOfY* and stored in *NewAssns*. Then the remaining def-use associations in *Triples* are assigned to *ValuePathAssns*. Finally, *DefUseAssns* is the union of *NewAssns* and *ValuePathAssns*.

```
function DeleteDefinition(S, Y)                                /* use the modified CFG */
   DefsOfY  = BackwardWalk(S,{Y})                              /* find statements with definitions of Y that reach S */
   Triples = ForwardWalk({(S,Y)}, false)                       /* find affected def-use associations */
   NewAssns  = {(Si,S′,Y): (S,S′,Y) ε Triples Λ Si ε DefsOfY}  /* form new def-use associations */
   ValuePathAssns = {(d,u,v: (d,u,v) ε Triples Λ d ≠ S}        /* find indirectly affected def-use associations */
   DefUseAssns= NewAssns ∪ ValuePathAssns                      /* combine new/affected def-use associations */
end DeleteDefinition
```

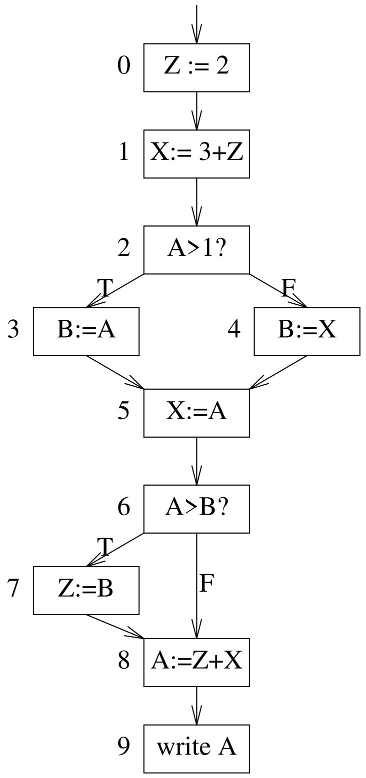
For example, if statement 5 in Figure 8 is deleted (Change4), *BackwardWalk* first identifies the definitions of Y that reach statement 5. The result is the definition of Y in statement 0. Next, *ForwardWalk* begins at statement 5 with Y as a reference and finds the affected value association (5,7,Y). Since the definition of Y at statement 5 has been deleted, definition 5 in def-use association (5,7,Y) is replaced by the definition of Y in statement 0 that was found by *BackwardWalk*. The changed def-use association is (0,7,Y). The def-use association (7,9,Z) is also found by *ForwardWalk* due to a change in the value of Z.

<u>*Change operator or value of a constant operand in statement* S: $Y := ..$</u>

If the operator or a constant operand used in an assignment statement is altered, no new def-use associations are created. However, the value of the variable defined by this altered statement changes, which causes value associations and possibly path associations. *OperatorChange* uses *ForwardWalk* to identify both value and path associations affected by this edit, and they are stored in *DefUseAssns*.

```
function OperatorChange(S)                          /* use the new CFG */
   DefUseAssns = ForwardWalk({(S,Y)}, false)        /* find all affected def-use associations */
end OperatorChange
```

To illustrate *OperatorChange*, consider Change1 in Figure 9; assume that there is an operator change in statement 1. *ForwardWalk* on statement 1 identifies the value associations that are affected by the change: (1,4,X), (4,7,B), (4,(6,7),B), (4,(6,8),B), (7,8,Z) and (8,9,A).



Change1: change operator in statement 1

ForwardWalk({(1,X)}, false): def-use associations to be retested are (1,4,X), (4,7,B), (4,(6,7),B), (7,8,Z), (8,9,A)

Change2: change condition in statement 6

BackwardWalk(6,{Z,B,X}): definition of Z in statement 0, definitions of B in statements 3 and 4, and definition of X in statement 5 reach statement 6 *ForwardWalk({(6,Z),(6,B),(6,X)}, true):* finds *Triples* (6,7,B), (7,8,Z) and (8,9,A), yielding def-use associations; def-use associations (3,7,B), (3,(6,7),B), (3,(6,8),B), (4,7,B), (4,(6,7),B), (4,(6,8),B), (7,8,Z) and (8,9,A)

Figure 9. Change1 modifies the operator in statement 1. *ForwardWalk* identifies the def-use associations to be retested. Change2 modifies the condition in statement 6. Both *BackwardWalk* and *ForwardWalk* are used to identify def-use associations affected by the change in statement 6.

Change operator or value of a constant operand in a branch condition in statement S

Changing the operators of a branch condition statement *S* creates no new def-use associations. Since the branch condition does not define any variable it does not directly affect the values of any def-use associations. (An example is changing a condition “X<Y” to “X>Y”.) However, all value and path associations influenced by the result of *S* must be identified. *ConditionChange* identifies the definitions that reach *S* using *BackwardWalk*, and stores them in *DefsOfV*. The uses of these definitions that are reachable from *S* and are control dependent on *S* are identified using *ForwardWalk*; the result is stored in *Triples*. *Triples* contains two types of def-use associations: (1) those def-use associations whose definition appears before *S* and whose use is control dependent on *S* and (2) those def-use associations whose definition is control dependent on *S*. Function *ConditionChange* computes those def-use associations of type (1) and stores them in *DefUseAssns*. Then, *ConditionChange* computes the def-use associations of type (2) and stores them in *ValuePathAssns*. The final set of affected def-use associations is found by adding the set of *ValuePathAssns* to *DefUseAssns*.

function *ConditionChange*(*S*) /* use the new CFG */
  DefsOfV = *BackwardWalk(S,V)* /* find statements with definitions of *V* that reach *S* */
  Triples = *ForwardWalk*({(S,V_i): *DefsOfV* contains a definition of V_i}, true) /* find value and path associations */
  DefUseAssns = {(S_i,S', V): (S,S',V) ε *Triples* Λ S_i ε *DefsOfV*} /* def-use associations of type (1) */
  ValuePathAssns = {(d,u,v): (d,u,v) ε Triples and d $\neq$ *S*} /* def-use associations of type (2) */
  DefUseAssns = *DefUseAssns* $\cup$ *ValuePathAssns* /* find all affected def-use associations */
end *ConditionChange*

Consider Figure 9 where there is a condition change in statement 6 (Change2). First, *BackwardWalk* on statement 6 identifies definitions that reach the changed predicate (i.e., definition of Z in statement 0, definitions of B in statements 3 and 4, and definition of X in statement 5). Then *ForwardWalk* is called with pairs {(6,Z),(6,B),(6,X)}, and the *Names* flag set. The *Triples* returned from this walk are used to form the def-use associations ((0,8,Z), (3,7,B), (3,(6,7),B), (3,(6,8),B), (4,7,B), (4,(6,7),B), (4,(6,8),B) and (5,8,X). Def-use associations (7,8,Z) and (8,9,A) are also affected and identified byt *ForwardWalk* for retesting.

Insert an edge (p,t) from statement p to statement t

Inserting an edge (*p,t*) can cause creation of new def-use associations since definitions may reach uses over the new paths created by (p, t). The variables defined by statements corresponding to uses of the new def-use associations may have their values affected. Thus, all def-use associations that depend on the values of affected variables must be identified. First, *InsertEdge* uses *BackwardWalk* to find all definitions that reach statement *t* in the altered control flow graph, *CFG′*; these definitions are stored in *DefsOfV*. Next, *ForwardWalk* is used to locate def-use associations in the old control flow graph, *CFG*, that depend on the definitions in *DefsOfV*; *Triples* is used to store these def-use associations. To identify the new and affected def-use associations involving the new edge (p, t), *ForwardWalk* is used to compute *Triples′* on *CFG′*. After removing the common def-use associations in *Triples* and *Triples′*, *Triples′* stores the new def-use associations, *NewAssns*. Then a *ForwardWalk* from *t* yields the value and path associations in *ValuePathAssns*. The set of affected def-use associations, *DefUseAssns*, is the union of *NewAssns* and *ValuePathAssns*. Since the addition of an edge modifies the control dependence information the modified flow graph must be reanalyzed to compute control dependences. Since the ForwardWalk is performed on both the old and the modified flow graphs, the control dependence information for both graphs must be made available.

action *InsertEdge*(p, t)
  DefsOfV = *BackwardWalk(t,{V})* on *CFG′* /* find definitions that reach t in new cfg */
  Triples = *ForwardWalk*$(\{(s, Def(s)): s \varepsilon DefsOfV\}, false)$ on *CFG*
  /* find def-use associations on old cfg */
  Triples′ = *ForwardWalk*$(\{(t, V_i)$: *DefsOfV* contains a definition of V_i and $V_i \neq Def(t)\}$, *true*) on *CFG′*
  /* find def-use associations on new cfg */
  Triples′ = $\{(s, t): s \varepsilon DefsOfV, Def(s)\ \varepsilon\ ref(t)\} \cup \{(s_i, s) : s_i\ \varepsilon\ DefsOfV$ and $(p, s, V_i)\ \varepsilon\ Triples'\}$
  NewAssns = *Triples′* - *Triples* /* find new def-use associations */
  ValuePathAssns = *ForwardWalk*$(\{(s, Def(s)): oppE\ (r, s)\ \varepsilon\ NewAssns\}, false)$ on *CFG′*
  /* find affected def-use associations */
  DefUseAssns = *NewAssns* $\cup$ *ValuePathAssns* /* all affected def-use associations */
end *InsertEdge*

To see the effects of inserting an edge in the control flow graph, consider Figure 10. In the figure, a partial original program and a modified version are given. Control flow graphs are shown for the original program (a), a modified version after an edge is deleted (b), and a modified version after an edge is added (c). (The general approach to translating higher level edits to lower level edits is discussed in Section 3.5.) First, *BackwardWalk* from statement 3 finds the definition of A in statement 1. Then *ForwardWalk* from statement 3 in the control flow graph for the original program (a) yields (1,5,A) and (5,6,X); *ForwardWalk* on the control flow graph for the new program (c) yields ((3,6,X). Thus, *NewAssns* contains (3,6,X), which must be tested.

Delete an edge (p,t) 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 deleted 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, *DeleteEdge* identifies the definitions, *DefsOfVp*, that reach statement p using *BackwardWalk*. Using this information the definitions, *DefsOfVt*, that reach t via the deleted edge are identified. Next *ForwardWalk* is used to locate def-use associations, *Triples*, in the old control flow graph, *CFG*, that depend on definitions in *DefsOfVt*. *ForwardWalk* on the new control flow graph, *CFG'*, is used to locate def-use associations, *Triples'*, that depend on definitions in *DefsOfVt* and are present in the modified control flow graph. By removing the def-use associations that have been eliminated due to edge deletion, we obtain the set of affected def-use associations, *DefUseAssns*. As was the case for edge addition, following edge deletion we must recompute control dependence information.

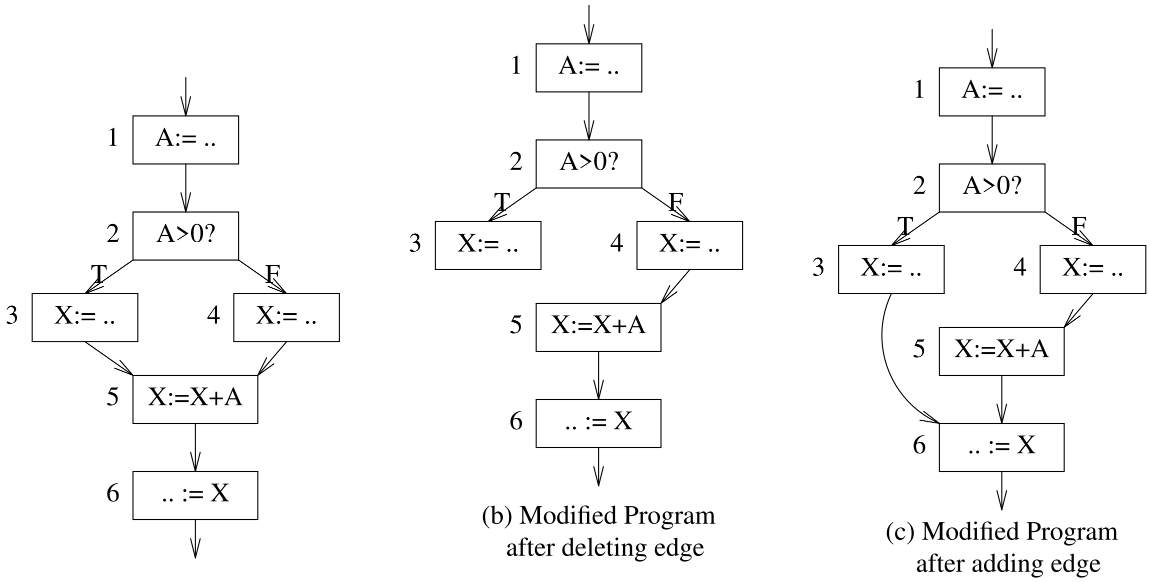
action *DeleteEdge*(p, t)
  DefsOfVp = *BackwardWalk(p,{V})* on *CFG* /* find definitions that reached p in old *CFG* */
  DefsOfVt = $\{p\} \cup \{n: n \in DefsOfVp$ and $Def(p) \neq Def(n)\}$ /* find definitions that reached t via p in old *CFG* */
  Triples = *ForwardWalk*$(\{(t, V_i)$: *DefsOfVt* contains a definition of V_i and $V_i \neq Def(t)\}$, *true*) on *CFG*
  Triples = $\{(s, t, Def(s)): s \varepsilon DefsOfVt, Def(s) \varepsilon\ ref(t)\} \cup \{(s_i, s) : s_i \varepsilon DefsOfVt$ and $(p,s,V_i)\ \varepsilon Triples\}$
  /* find old def-use associations through (p, t) */
  Triples′ = *ForwardWalk*$(\{(s, Def(s)): s \varepsilon DefsOfVt\}, false)$ on *CFG′*
  /* find new def-use associations in *CFG′* */
  DefUseAssns = *Triples* - (*Triples* – *Triples′*) /* set of affected def-use associations */
end *DeleteEdge*

Original Program (partial)

```
    . .
    A := ..
    if A > 0 then
        X := . .
        goto L
    else X := . .
L:  X := X + A
L′: .. := X
    . .
```

Modified Program (partial)

```
    . .
    A := . .
    if A > 0 then
        X := . .
        goto L′
    else X := . .
L:  X := X + A
L′: . . := X
    . .
```



(a) Original Program

(b) Modified Program after deleting edge

(1,5,A) and (5,6,X) must be retested

(c) Modified Program after adding edge

(3,6,X) must be tested

Figure 10. A partial program and its modified version. The control flow graph for the original program is shown on the left, and control flow graphs for the modified version are shown in the center and on the right.

To see the effects of deleting an edge in the control flow graph, consider Figure 10. *BackwardWalk* from statement 3 determines that the definition of variable A in statement 1 reaches statement 5 via the deleted edge (3,5). Then the *ForwardWalk* from statement 5 yields def-use associations (1,5,A) and (5,6,X). Since these def-use associations are not deleted in the modified program, both must be tested.

3.5. Translating High Level Edits to Low Level Edits

The approach to translating higher level edits into lower level edits for processing by our algorithms is:

a) make structural changes to the control flow graph;

b) add/modify assignment statements;

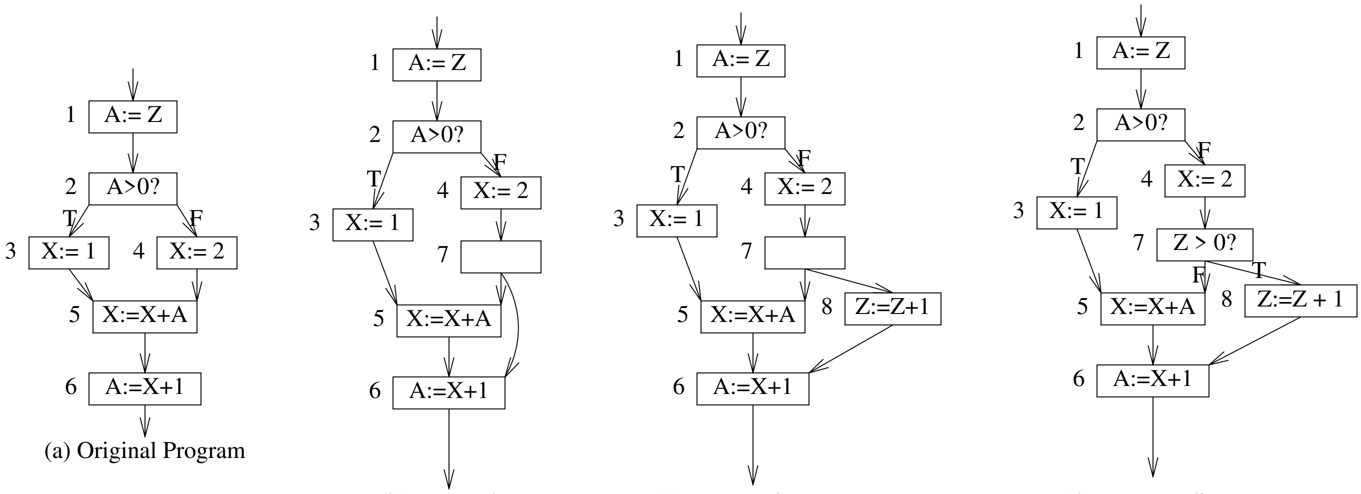
c) add/modify predicates.

Original Program Segment

```
. .
A := Z
if A > 0 then
    X := 1
else
    X := 2
X := X + A
A := X + 1
. .
```

Modified Program Segment

```
. .
   A := Z
   if A > 0 then
       X := 1
   else
       X := 2
       if Z > 0 then
          Z := Z + 1
          goto L
   X := X + A
L: A := X + 1
. .
```



(a) Original Program

(b) Insert edge

(c) Insert assignment statement

(d) Insert predicate

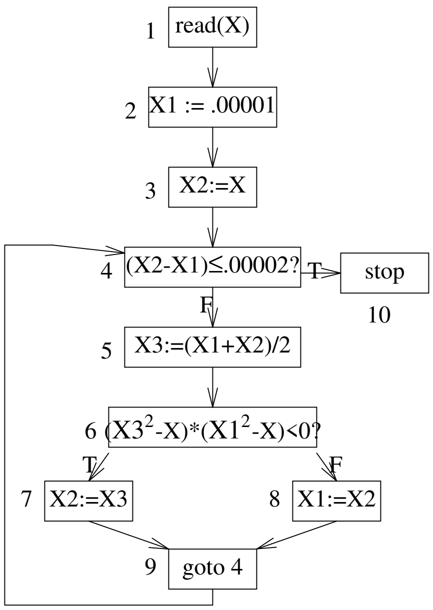
Figure 11. A partial program on the left and the modified versions that result in a predicate change in the program.

Figure 11 shows an example of inserting a predicate statement. In the figure, both the original program segment and the modified program segment are shown. The control flow graph in (a) represents the original program segment. The control flow graph in (b) shows the insertion of the edge to the target of the conditional branch. Then, in (c), the body of the conditional statment is inserted; here only an assignment statement is inserted. Finally in (d), the predicate statement is added to the control flow graph. In each step, the algorithms presented in this paper are used to process the edit and identify those def-use associations to (re)test.

After each low level edit, the def-use associations that must be retested are identified. After all low level edits have been processed the set of all possible def-use associations that may require retesting are known. Since the later low level edits may cause some of the previously identified def-use associations to be invalidated, a single pass over the def-use associations is required to identify and remove these def-use associations. The backward walk algorithm is used to determine the validity of the def-use associations. The invalid associations are discarded and the remaining def-use associations are now tested. It is important to note that although a high level edit may translate into several low level edits, testing is not performed after each low level edit but rather it is performed once per high level edit.

4. Testing Affected Def-Use Associations

Next, the regression testing technique is illustrated using an example. Consider the program whose control flow graph is shown in Figure 12. The program should compute the square root for input variable X by repeatedly halving the interval between the two estimates, X1 and X2. The table on the right in Figure 12 gives the def-use associations for the program that are required to satisfy the all-uses data flow testing criterion. The program has



Def-use Information

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

Figure 12. Program to compute the square root of X that contains an error in statement 8 that must be changed to “X1:=X3”.

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. The sequel shows how the algorithms handle the changes.

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

Because X2 is deleted, ForwardWalk first finds the uses of X1 in statements 4, 5 and 6. Thus, detected def-use associations for X1 are (8,(4,6),X1), (8,(4,5),X1), (8,5,X1), (8,(6,7),X1) and (8,(6,8),X1). Since the definition of X3 in statement 5 may be affected, def-use associations for X3, (5,(6,7),X3), (5,(6,8),X3) and (5,7,X3) are detected. The use of X3 in statement 7 may affect the definition of X2 and def-use associations for X2 (7,(4,5),X2), (7,(4,10),X2) and (7,5,X2) are identified. Although the definition of X3 in statement 5 may be affected by this use of X2, the processing stops since def-use associations 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 association (5,8,X3) 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 associations are returned. Figure 13 lists the def-use associations that are affected after the program change and must be tested to satisfy the all-uses criterion.

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

Figure 13: Def-use Associations for Regression Testing

Although this paper has described the use of the sliced-based algorithms to satisfy the all-uses criterion, the technique is applicable to the other data flow testing criteria. For a given change, these algorithms identify the directly and indirectly affected def-use associations in the program. Thus, these are the only def-use associations that must be considered for adequacy when retesting the program. If the user desires all-du-paths coverage instead of all-uses, then instead of finding a test that executes some subpath from the definition to the use, tests would be required to execute all du-paths between the definition and the use. If the all-defs coverage is desired, then instead of satisfying all def-use associations, only one of them would be required for each definition.

After the affected def-use associations are identified (see Figure 13), tests must be executed to exercise these def-use associations. One technique is to maintain a test suite that is used to retest the program. If a test suite is maintained, tests are selected from the the test suite to retest the def-use associations 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 used to test that pair. Information about the test includes the input values and the execution path. The test suite is examined to determine the tests that satisfy the affected def-use associations and the changed code. These tests are rerun and we check to see which affected associations are satisfied by a test, as well as updating the test suite. If a def-use pair does not have an associated test, then a new test can be generated. This technique requires fewer tests than previous methods since we do not execute all tests that traverse the changed node. For the example in Figure 13 the def-use associations due to statements 1, 2, and 3 are not tested following the program change.

To save the overhead involved in storing and updating test suites, another approach is to generate tests only for the changed parts of the program during regression testing. Since the sliced-based algorithms explicitly identify both directly and indirectly affected def-use associations, the only tests that must be run are those that satisfy these associations and all-uses data flow testing coverage of the program will still be provided. Thus, unlike other regression testing techniques, the sliced-based method identifies affected def-use associations without the overhead of maintaining and updating a test suite.

In either of the scenarios given above, new tests may be required. Tests can be generated manually by the user or automatically using a test generator. When a new test is required, a program slice can be computed based on the affected def-use associations to assist in generation of tests. This program slice contains those statements that are needed to traverse all affected and untested def-use associations. If tests are generated manually, the program slice enables the user to focus on the program statements pertinent to the affected def-use associations. If the tests are automatically generated, the test generator only considers the program statements that are in the slice. In either case, using the slice reduces the test generation effort since only the input variables on the slice are considered. Additional details of our slicing algorithm and its use in test generation can be found in references [7, 8].

Another approach to regression testing based on slicing was developed by Bates and Horwitz[2]. This approach uses backward slicing algorithms on program dependence graphs to determine the affected parts of a program that require retesting. Following program changes the program dependence graph must be reconstructed before slicing algorithms can be applied for identifying the affected parts of the program. This construction of a program dependence graph requires the exhaustive recomputation of all def-use associations in the program or the incremental update of changed def-use associations based on previous def-use associations. A different approach to regression testing, also based on slicing, was developed by Rothermel and Harrold [17, 18]. This approach also uses a program dependence graph representation to identify the changed def-use pairs for regression testing. Unlike the above approaches to slicing, the approach presented in this paper only requires partial data flow analysis following program changes and does not depend on any def-use history. Rothermel and Harrold also consider interprocedural regression testing while in this paper we focus on intraprocedural testing. However, since the slicing algorithms presented in this paper are based on Weiser's technique [20] which handles interprocedural slicing, these algorithms can also be extended to allow interprocedural regression testing.

The problem of performing partial data flow analysis is also addressed by a demand driven data flow framework described in Reference[4]. However, this framework does not apply to analyses that require consideration of control dependence information. Therefore, although the framework could be used to derive the *BackwardWalk* algorithm, it cannot be used in developing the *ForwardWalk* algorithm.

5. Conclusion

This paper has a regression testing technique that utilizes slicing. This technique identifies both directly and indirectly affected def-use associations. The use of the slice for regression testing is efficient in terms of both memory and time overhead. Unlike previous regression techniques, this approach neither needs to completely recompute data flow information after a change nor maintain a history of previous data flow computation for incremental updates. Instead, the approach recomputes the partial data flow that is needed, as driven by the program changes. Also, the approach does not need the overhead of maintaining a test suite, which includes the input, output and updates of the test suite. If the test suite is maintained, the approach reduces the number of tests that must be rerun to provide full testing coverage and to update the test suite. Although the technique has been presented to satisfy the all-uses criterion, it could easily be modified for other data flow testing criteria. Using interprocedural slicing, the technique can also be extended to interprocedural regression testing.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman, in *Compilers, Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, Massachusetts, 1986.
2. S. Bates and S. Horwitz, "Incremental program testing using program dependence graphs," *Conference Record of 20st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 384-396, January 1993.
3. 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.
4. E. Duesterwald, R. Gupta, and M.L. Soffa, "Interprocedural data flow analysis on demand," *Proceedings of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 37-48, January 1995.
5. J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319-349, July 1987.
6. 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.
7. R. Gupta and M. L. Soffa, "Employing static information in the generation of test cases ," *Journal of Software Testing, Verification and Reliability*, vol. 3, no. 1, pp. 29-48, December 1993.
8. R. Gupta and M. L. Soffa, "A framework for partial data flow analysis," *Proceedings of International Conference on Software Maintenance*, pp. 4-13, September 1994.
9. M. J. Harrold and M. L. Soffa, "An incremental approach to unit testing during maintenance," *Proceedings of the International Conference on Software Maintenance*, pp. 362-367, October 1988.
10. M. J. Harrold, "An approach to incremental testing," Technical Report 89-1 Department of Computer Science, Ph.D. Thesis, University of Pittsburgh, January 1989.
11. S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 1, pp. 26-60, January 1990.
12. B. Korel and J. Laski, "A tool for data flow oriented program testing," *ACM Softfair Proceedings*, pp. 35-37, December 1985.
13. S. C. Ntafos, "An evaluation of required element testing strategies," *Proceedings of 7th International Conference on Software Engineering*, pp. 250-256, March 1984.

14. T. J. Ostrand and E. J. Weyuker, ‘‘Using data flow analysis for regression testing,’’ *Proceedings of Sixth Annual Pacific Northwest Software Quality Conference*, pp. 58-71, September 1988.

15. A. Podgurski and L. Clarke, ‘‘A formal model of program dependences and its implications for software testing, debugging, and maintenance,’’ *IEEE Transactions on Software Engineering*, vol. 16, no. 9, pp. 965-979, September 1990.

16. S. Rapps and E. J. Weyuker, ‘‘Selecting software test data using data flow information,’’ *IEEE Transactions on Software Engineering*, vol. SE-11, no. 4, pp. 367-375, April 1985.

17. G. Rothermel and M.J. Harrold, ‘‘A safe, efficient algorithm for regression test selection,’’ *Proceedings of the International Conference on Software maintenance* , pp. 358-367, September 1993.

18. G. Rothermel and M.J. Harrold, ‘‘Selecting tests and identifying test coverage requirements for modified software,’’ *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 169-184, August 1994.

19. 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.

20. M. Weiser, ‘‘Program slicing,’’ *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 352-357, July 1984.