

Interprocedural Data Flow Testing*

Mary Jean Harrold and Mary Lou Soffa
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260

ABSTRACT

As current trends in programming encourage a high degree of modularity, the number of procedure calls and returns executed in a module continues to grow. This increase in procedures mandates the efficient testing of the interactions among procedures. In this paper, we extend the utility of data flow testing to include the testing of data dependencies that exist across procedure boundaries. An interprocedural data flow analysis algorithm is first presented that enables the efficient computation of information detailing the locations of definitions and uses needed by an interprocedural data flow tester. To utilize this information, a technique to guide the selection and execution of test cases, that takes into account the various associations of names with definitions and uses across procedures, is also presented. The resulting interprocedural data flow tester handles global variables, reference parameters and recursive procedure calls, and is compatible with the current intraprocedural data flow testing techniques. The testing tool has been implemented on a Sun 3/50 Workstation.

1. INTRODUCTION

Although a number of data flow testing methodologies have been developed and studied in recent years,[8, 11, 14, 15, 18, 19] their utility has been restricted to testing data dependencies that exist within a procedure (i.e., *intraprocedural*). Testing the data dependencies that exist among procedures (i.e., *interprocedural*) requires information about the flow of data across procedure boundaries, including both calls and returns. The data dependencies that exist between procedures both *directly* over

single calls and returns and *indirectly* over multiple calls and returns are needed. The current data flow testing tools either use intraprocedural data flow analysis typically employed in compiler optimization to determine the data dependencies or determine the definition-use pairs from the source code by building and then searching the program's def-use graph. Although interprocedural data flow analysis algorithms do exist, [2-7, 9, 17] they do not provide the detailed information (i.e., the locations of definitions and uses that reach across both procedure calls and returns) needed for the interprocedural data flow testing. Also, the methods for guiding the actual data flow testing do not currently handle the renaming of variables that is required when performing interprocedural testing.

The underlying premise of all of the data flow testing criteria is that confidence in the correctness of a variable assignment at a point in a program is dependent on whether some test data has caused execution of a path from the assignment (i.e., *definition*) to points where the variable's value is used (i.e., *use*). Test data adequacy criteria are used to select particular definition-use pairs or subpaths that are identified as the test case requirements for a program. Then, test cases are generated that satisfy the requirements when used in a program's execution. Thus, interprocedural data flow testing consists of (1) determining the definition-use information for definitions that reach across procedure boundaries (both calls and returns) to meet the adequacy criteria and (2) guiding the selection and execution of test cases that meet the requirements. The problems of determining interprocedural definition-use information include the development of an efficient technique that is procedure call site specific and handles reference parameters, global variables and recursion for direct and indirect data dependencies. Direct dependencies exist when either (1) a definition of an actual parameter in one procedure reaches a use of the corresponding formal parameter in a called procedure or (2) a definition of a formal parameter in a called procedure reaches a use of the corresponding actual parameter in the calling procedure. Conditions for indirect dependencies are similar to direct

* This work was partially supported by the National Science Foundation under Grants CCR-88001104 to the University of Pittsburgh.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1989 ACM 089791-342-6/89/0012/0158 \$1.50

dependencies except that multiple levels of procedure calls and returns are considered. When a formal parameter is passed as an actual parameter at a call site, an indirect data flow dependency may exist. In this case, a definition of an actual parameter in one procedure may have uses in procedures more than one level away in the calling sequence, considering both calls and returns. For the selection and execution of the test cases, the presence of reference parameters requires incorporating the renaming of variables as procedures are called and returned, which complicates the design of a testing tool.

This paper extends data flow testing to include *interprocedural data flow testing* by developing both an efficient interprocedural data flow analysis technique that gathers the necessary information about interprocedural data dependencies and a system that uses the information to guide the interprocedural testing. Through this extension, data flow testing can be used to test individual procedures in a module more accurately since data flow information about called procedures is known as well as the interactions of these procedures as indicated by the interprocedural data dependencies. In particular, we have designed an interprocedural data flow testing tool that first computes the required interprocedural definition-use information, for both direct and indirect dependencies, and then uses it to provide integration testing of the procedures in a module. For the interprocedural data flow analysis, our technique summarizes the procedure information at call sites and then propagates this information throughout the module to obtain the interprocedural definition-use information for both global variables and reference parameters. Our technique handles recursion and thus permits data flow analysis of recursive procedures. By computing the interprocedural data dependencies in an efficient way prior to testing, existing path selection techniques based on data flow [11, 19] can be used for interprocedural testing. To guide the testing, the system recognizes the calls to, and returns from, procedures and handles the associations of various names with a definition as the execution path is being inspected.

With our interprocedural data flow testing tool, integration testing can be performed in several ways. One is to use 'incremental bottom-up' testing of the procedures in the module. As each procedure is tested, it is also integrated with any procedures that it calls directly or indirectly. Thus, the test case requirements include testing definitions that have uses within the procedure, definitions that have uses in a called procedure, and definitions in a called procedure that reach uses in a calling procedure. Another way is to use the 'big bang' approach which first tests each procedure in isolation and then integrates them all at once by providing test cases that test only those definitions that have interprocedural uses. In either case, the direct and indirect data dependencies are used to compute the required definition-use information for the testing.

Section 2 introduces an example to illustrate some of the problems that are inherent to interprocedural data flow testing. In Section 3, the issues involved in the design of the system are discussed. Section 4 describes the interprocedural data flow testing system. Our conclusions are given in Section 5.

2. MOTIVATING EXAMPLE

To illustrate some of the problems involved in interprocedural data flow testing, we consider an example. Our data flow tester uses the data flow testing methodology proposed by Rapps and Weyuker [19] and later extended by Frankl and Weyuker [11]. One criterion, 'all-uses' requires that each definition be tested to all of its uses. If the use is in a computation statement (e.g., assignment), testing is from the block containing the definition to the block containing the use. If the use is in a predicate (e.g., conditional), testing is from the block containing the definition to the successors of the block containing the use. This ensures that both edges from a conditional statement containing a use are tested. The subpath traversed in the testing must contain no redefinition of the variable (i.e., it must be a *definition-clear* subpath). To illustrate, consider the module Main and the flow graph for procedure GetMax which are given in Figure 1. GetMax is a recursive procedure that inputs the index of the first and last elements of a list of integers and returns the maximum value in the list. S is a global array initialized in Main, that stores the lists of integers for which the maximum (MX) is to be found. To simplify the example, only the definitions and uses of reference parameters that reach across procedure boundaries are considered. These include definitions and uses of the formal reference parameter for GetMax (i.e., MX) and actual parameters at the call sites that are bound to formal reference parameters in called procedures (i.e., MX, M1 and M2). The definition-use pairs required to test GetMax are identified from the graph and given in Table 1.

<i>variable</i>	<i>definition</i>	<i>use</i>
MX	B3	B8
	B7	B8
M1	B5	B7
M2	B6	B7

Table 1: Definition-Use Pairs for GetMax

Consider a test case whose elements for S are 3,5,1,6. When the procedure is executed with this test data as input, the path traversed through the basic blocks is 1,2,4,5,1,2,3,8,6,1,2,3,8,7,8. Comparison of this test path with the required definition-use pairs reveals that it satisfies all of the test case requirements. However, with this single

```

module Main
declare
  S: an array 1...N of integer;
  I,MAX,MIN: integer;
begin
  for I := 1 to N do read(S[I]);
  GetMax(1,N,MAX);
  write(MAX);
end;

procedure GetMax;
input
  F,L: integer;
  MX: reference integer;
declare M1,M2,MD: integer;
begin
  if F+1=L then PairMax(S[F],S[L],MX)
  else begin
    MD := (F+L) DIV 2;
    GetMax(F,MD,M1);
    GetMax(MD+1,L,M2);
    PairMax(M1,M2,MX);
  endif;
end;

procedure PairMax;
input I,J,K: reference integer;
begin
  if I>J then K := I
  else K := J;
end;

```

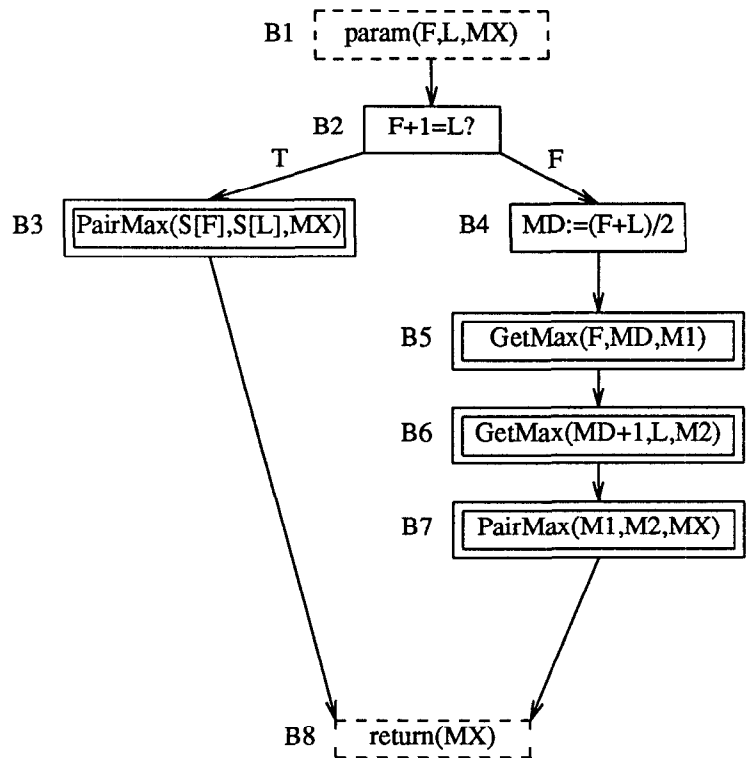


Figure 1: Example Program and Flow Graph for Procedure GetMax

test case, several interprocedural definition-use pairs remain untested. For example, there are two definitions of K in procedure PairMax and both of them reach the use of MX in B8 but the test case tests only one of them. To test the integration of procedures, information about the locations of uses of definitions that reach across procedure boundaries is required. With this information, definition-use pairs include uses of a definition that occur as a result of procedure calls. This allows better selection of test data to test the integration of procedures.

In addition to the direct dependencies that exist between definitions and uses in GetMax and PairMax that are discussed above, indirect dependencies exist between Main and PairMax. For example, since formal parameter K in PairMax is returned to GetMax and subsequently to Main, there is a data dependency between the definitions of K in PairMax and the use of MAX in Main. Determining the indirect dependencies is accomplished by consid-

ering the possible uses of definitions along the calling sequences. For efficiency, this information must be computed prior to the testing by considering local information and propagating it throughout the module instead of computing it during testing.

Assuming that the interprocedural data dependencies have been computed, another problem involves guiding the testing of the required interprocedural definition-use pairs. Consider the definition of K in PairMax and its use of MAX in Main. To determine the definition-clear subpath from the definition to the use of a variable requires traversal through several procedures where the name of the variable changes. Specifically, in this example, K in PairMax, MX in GetMax, and MAX in Main are all names for the same memory location. Thus, access to information about the binding of actual and formal parameters at call and return sites is required.

3. DESIGN ISSUES

The design of an interprocedural data flow tester includes issues concerned with both a technique to compute the interprocedural data dependencies and a method to guide the testing process. This section discusses these issues.

Existing interprocedural data flow analysis techniques [2-7, 9, 17] vary in the type of the information provided and the efficiency of gathering the information. Most of the techniques concentrate on providing sets of variables that are used or modified by procedure calls using either flow sensitive or flow insensitive information.† Although this information is useful in optimizations and parallelization, it does not provide the locations of the definitions and uses of variables that reach across procedure boundaries. Thus, to determine the test case requirements for interprocedural data flow testing, a new method for computing the interprocedural definition-use information is required. This involves the development of an efficient representation of the procedures within a module and an algorithm to propagate data flow information throughout a module, taking into account reference parameters, global variables and recursive procedure calls.

One possible way to represent the program is by *in-line substitution* of procedures at call sites. In addition to the obvious problem of the memory requirements, in-line substitution has other inherent problems. Both scoping of local variables in procedures and binding of formal and actual parameters are difficult because the entire module is viewed as one procedure. Additionally, recursive procedures cannot be represented. Another possible representation is the traditional *call graph* of a module where nodes in the graph represent procedures and edges represent call sites. However, the call graph is not sufficient for computing the definition-use information across procedure boundaries because it has no return information and provides no information about the control flow in individual procedures. Other representations, such as the *program summary graph*[5] and the *super graph*[17] provide this return and control flow information. With the super graph, each call to a procedure accesses a single copy of the procedure. This eliminates some of the memory problems of in-line substitution but still may be prohibitive for large programs. The program summary graph summarizes some of the required information at call sites but this information does not indicate the locations of definitions and uses that reach across procedure boundaries.

† A technique is *flow sensitive* if it incorporates information about the flow of control in the called procedures and *flow insensitive* otherwise.

Closely related to the choice of the module representation is the design of an algorithm that propagates the summarized information throughout the module. Here, the requirements are that the algorithm be efficient in both memory requirements and execution time, and that it handles interprocedural definition-use computation for both recursive and nonrecursive procedures. Definitions that reach across procedure boundaries include definitions of global variables that reach a call or return site, definitions of actual parameters that reach a call site and definitions of formal parameters that reach a return site. A similar situation exists for uses. Most of the existing interprocedural data flow analysis techniques make worst case assumptions at the call sites that involve recursion due to the fact that incomplete information is known about the called procedure. Thus, new techniques are required to handle these problems.

Other problems deal with guiding the testing of a module to meet the test case requirements that were computed using the results of the interprocedural data flow analysis. If only global variables are to be tested, a tool similar to that employed for intraprocedural data flow testing [10] can be used. A procedure is instrumented to record the execution path that occurs when the procedure is executed with particular test data as input. The tool then searches the execution path for the desired definition and use, making sure that the variable is not redefined on the subpath between them. Thus, an important component of this tool is the information about the locations of all definitions of a variable being considered. With global variables, the names of the variables remain the same throughout the module and thus, the locations of the definitions can be easily computed during interprocedural analysis and used in the testing.

However, for reference parameters, this is not the case. While searching for a definition and use pair, the execution path moves from procedure to procedure, causing the name of a definition to change. In order to ensure that the subpath from the definition to the use has no redefinition of the associated variable, the pairings of the actual and formal parameters must be handled when the execution path reaches a call or return site. A problem occurs when a definition and use are separated by a number of procedure calls. The actual parameter associated with a use in a procedure must be bound to the appropriate formal parameter on the returns along the call chain.

4. INTERPROCEDURAL DATA FLOW TESTING

The interprocedural data flow testing is performed in two parts: (1) static analysis of the module to compute the interprocedural definition-use information for the test case requirements and (2) dynamic testing that guides the testing of the module to meet the requirements. Sections

4.1 and 4.2 describe our interprocedural tester. To simplify our discussion, we assume that the user has chosen the ‘all-uses’ criterion for the testing and that only the definitions of reference parameters that have interprocedural uses are considered. Global variables can be handled similarly.

4.1. Computing the Interprocedural Data Flow Information

Our technique to compute the interprocedural definition-use information[12] represents the module by a graph, the *interprocedural flow graph* (IFG), that is based on the program summary graph. [5] Our algorithm computes the interprocedural definition-use information using the IFG for modules with both recursive and nonrecursive procedures. The technique has four steps.

Step 1: Construction of IFG subgraphs to abstract control flow information for each procedure in the program. A subgraph is constructed for each procedure where nodes represent regions of code associated with points that are of interest interprocedurally, and edges represent the control flow in the procedure. Local information is computed for non-local variables and is attached to appropriate nodes in the graph.

Step 2: Construction of an IFG to represent the interprocedural control flow in the program. The subgraphs of the procedures, obtained in step 1, are combined to create the IFG which is constructed by creating edges that represent the bindings of formal and actual parameters in both called and calling procedures. Preserved information is computed for each procedure, using the IFG, and edges that represent this information are added to the graph.

Step 3: Propagation throughout the graph to obtain global information. The local information at each node is propagated in two phases throughout the graph resulting in the interprocedural definitions that reach, and the interprocedural uses that can be reached from, the parts of the program represented by the node in the graph.

Step 4: Computation of the interprocedural def-use and use-def chains. Interprocedural def-use and use-def chains are computed using both the local information and the propagated global information.

4.1.1. Steps 1 and 2: Constructing the Interprocedural Flow Graph

As procedures are processed one at a time in any order, the results of the intraprocedural data flow analysis are used to construct the IFG. The IFG has four types of nodes: entry, exit, call and return. Entry and exit nodes

represent procedure entry and procedure exit respectively. Both nodes are created for every formal reference parameter of every procedure. Call and return nodes represent procedure invocation and procedure return respectively and both are created for every actual parameter of every call site. Edges from call nodes to entry nodes and exit nodes to return nodes correspond to the binding of formal and actual parameters. Reaching edges from entry and return nodes to call and exit nodes summarize the control information in the procedure by indicating that a definition that reaches the source of an edge also reaches the sink of the edge. This reaching edge is strictly intraprocedural since it is computed without incorporating the control structure of called procedures by using ‘best case’ assumptions at call sites. (Best case is to assume that there is no definition or use of the variable in the called procedure and that the variable is not preserved over the call). Inter-reaching edges from call nodes to return nodes that abstract the reaching information of the called procedure allow this reaching information to be incorporated into the calling procedure. This edge is added to the IFG if the variable is preserved[16] over a call to the procedure. The inter-reaching edges allow the calling context of the called procedures to be preserved during the propagation.

Figure 2 gives the partial IFG for the module in Figure 1. Here, we show the part of the graph that represents reference parameter MX in procedure GetMax and the reference parameters for procedure PairMax at the call sites in B5, B6 and B7. The rest of the IFG is similar. In the graph, circles represent call and return nodes, double circles represent entry and exit nodes, solid lines correspond to binding edges, dashed lines to reaching edges and bold lines to inter-reaching edges. Thus, nodes 3, 5, 7, 9 and 11 are call nodes while nodes 4, 6, 8, 10 and 12 are their corresponding return nodes respectively. Entry and exit pairs are nodes {1,2}, {13,14}, {15,16}, and {17,18}. Reaching edge (1,11) indicates that a definition of MX that reaches the entry to GetMax also reaches the call to PairMax where it is used as a parameter. Likewise, the reaching edge (12,2) indicates that a definition of MX that reaches the return from PairMax also reaches the exit from GetMax. The flow of control in PairMax for formal reference parameters I and J is summarized by reaching edges (13,14) and (15,16). The last set of reaching edges, (4,7) and (6,9) indicate that a definition of the parameter that reaches the return from one procedure subsequently reaches another call site where it is used as a parameter. Finally, inter-reaching edges (7,8) and (9,10) indicate that definitions of the actual parameter that reach the call to PairMax are still available on the return from PairMax.

In step 1, as procedures are processed and IFG nodes are created, definition and use information about

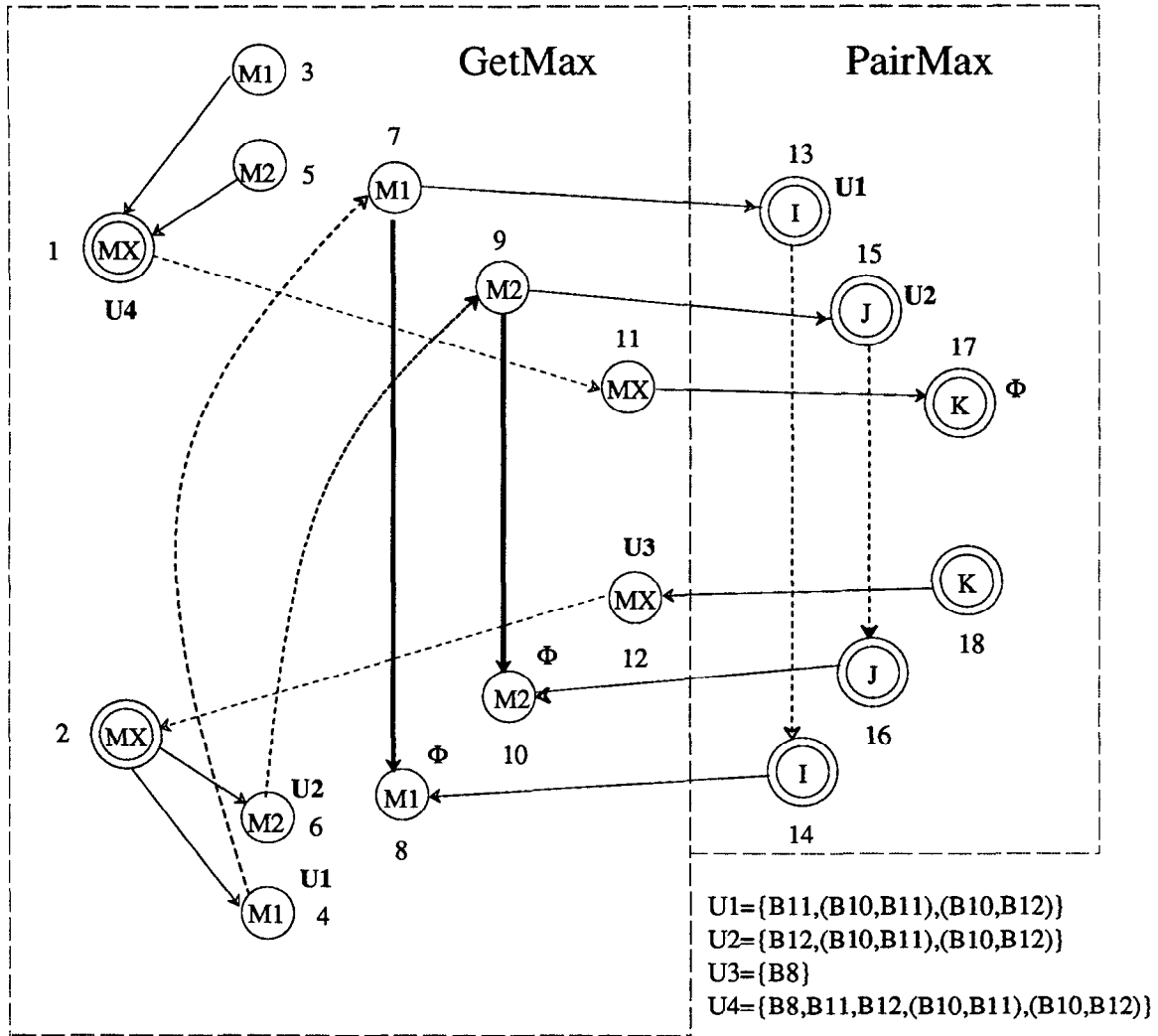


Figure 2: Partial IFG for Module in Figure 1

formal and actual parameters in a module is recorded. Sets containing information about definitions (i.e., DEF) and uses (i.e., UPEXP) are attached to the nodes in the IFG. These sets contain data flow information about a procedure. They are roughly analogous to the 'gen' and 'use' sets that are computed during intraprocedural data flow analysis for each basic block in the flow graph of a procedure[1] (i.e., they represent local information about certain parts of the module that is propagated throughout the graph). We limit our discussion to the computation of the UPEXP sets. Analogous techniques are used to compute the DEF sets. The UPEXP sets are attached to entry and return nodes. The UPEXP of an entry node is the set of uses of the formal reference

parameter that can be reached from the beginning of the procedure; the UPEXP of a return node is the set of uses of the actual reference parameter that can be reached from the return from a procedure. To compute the UPEXP set of a procedure, the control flow graph of the procedure is constructed and intraprocedural data flow analysis is performed. Again, consider the IFG in Figure 2 where the UPEXP sets (i.e., $U1, \dots, U4$) are attached to the entry and call nodes. Consider entry node 13 in the IFG which represents the entry into procedure PairMax for formal reference parameter I. In the procedure, I is used in B11 and on edges (B10,B11) and (B10,B12) (refer to Figure 3). Thus, the UPEXP set attached to node 13 is $\{B11, (B10, B11), (B10, B12)\}$. Another example is entry

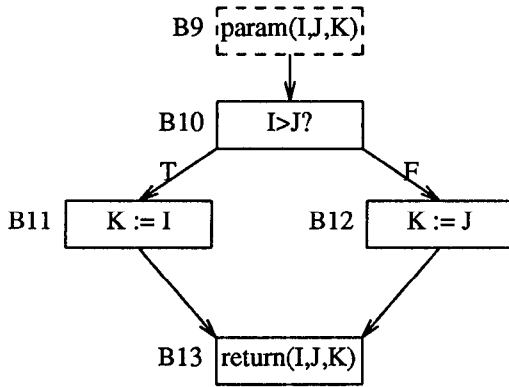


Figure 3: Flow Graph for PairMax

node 1 which represents the entry into procedure GetMax for formal reference parameter MX. MX has a local use in B8 and thus, UPEXP, attached to node 1, contains B8.

During this first step, the intraprocedural data flow information is also used to determine the existence of definition-clear subpaths that are required to create the reaching edges. Since there is a definition-clear subpath with respect to MX from the beginning of GetMax to the call to PairMax in B7, reaching edge (1,11) is created. Another reaching edge is (13,14) which represents the fact that there is a definition-clear subpath with respect to variable I from the beginning to the end of PairMax.

After all procedures have been processed, step 2 consists of constructing the IFG by creating the appropriate binding edges among actual and formal parameters in the procedures. In Figure 2, call binding edges (7,13), (9,15), (11,17) and return binding edges (14,8), (16,10), (18,12) are added at this time. The inter-reaching edges are obtained by processing the graph using an iterative algorithm[5] to determine whether, for each entry node, the formal parameter is preserved. Inter-reaching edges are created for each call-return pair whose associated entry node is preserved. In Figure 2, edges (7,8) and (9,10) are added in step 2.

4.1.2. Step 3: Computing the Reaching Definitions and Reachable Uses

The third step consists of propagating the local definitions throughout the module using the IFG to obtain the sets of interprocedural reaching definitions (IN_D and OUT_D) and reachable uses (IN_U and OUT_U) for each node in the graph. For reaching definitions, the algorithm propagates the DEF sets forward in the graph as far as they reach. Likewise, the algorithm propagates the UPEXP sets in the graph as far as they can be reached except the

flow is in the backward direction in the graph. To preserve the calling context, propagation is restricted over certain edges in the graph. For example, propagation of the UPEXP sets is restricted to the reaching edges, the inter-reaching edges and the return binding edges. The call binding edges are excluded to preserve the calling context of the called procedures.

To illustrate, consider the results of the computation of the OUT_U of node 18 in Figure 2. Since there is a path from this node to node 13 (i.e., 18,12,2,4,7,13), the UPEXP set of node 13 is propagated throughout the graph and added to the OUT_U of node 18. Other nodes with UPEXP sets reachable from node 18 are nodes 15, 9, 6, 4, 7 and 13. The UPEXP sets of all of these nodes are added to the OUT_U of node 18. Thus, the OUT_U of node 13 is the set {B8,B11,B12,(B10,B11),(B10,B12)} which represents the uses of parameter I reachable from the definition of K in the call to PairMax.

4.1.3. Step 4: Computation of the Interprocedural Def-Use and Use-Def Chains

With the OUT_U attached to all nodes in the graph, the interprocedural def-use pairs can be computed. In procedure PairMax, there are definitions of K in B11 and B12 which reach the associated exit node 18 in the IFG. Thus, the uses that can be reached from this definition are those in the OUT_U of node 18. These are the uses of I, J and MX in B8, B10, B11 and B12. Thus, we get the following definition-use associations for testing.

variable	definition	uses
K in PairMax	B11	B8,B11,B12,(B10,B11), (B10,B12)
	B12	B8,B11,B12,(B10,B11), (B10,B12)

4.1.4. Complexity Analysis

The IFG inherits its space requirements from the program summary graph whose size is proportional to the length of the program.[5] The time complexity of the algorithm is determined by considering each of the four steps. Clearly, in the first step, the creation of the graph requires one visit to each of the n nodes in the IFG. The last step in the algorithm is performed by considering the definition in each DEF set and combining the appropriate OUT_U sets to get the interprocedural def-use chains. This step also requires one visit to each node during the computation. In step two the preserved information that is required for the inter-reaching edges is computed. For modules with no recursion, this computation is linear in the number of nodes in the IFG. The propagation of the

```

algorithm Accept
input
  TRACEFILE: file of block numbers of execution path;
  VAR: variable being defined;
  DBLK: block number of definition;
  UBLK1: block number of c-use (or source of edge of p-use);
  UBLK2: block number of sink of edge of p-use;
  TY: c-use or p-use;
declare
  B: block number;
  DEFSET: blocks in current procedure where VAR is defined;
  PROCSTACK: stack to store call chain;
  CONTINUE: boolean;
begin
  ACCEPT := false; PROCSTACK := NULL;
  DEFSET := SetOfBlocks(VAR);
  while not eof(TRACEFILE) or not ACCEPT do
    repeat
      B := GetNextBlock(TRACEFILE);
      if IsACall(B) then Push(PROCSTACK,B)
      elseif IsAReturn(B) then Pop(PROCSTACK);
    until eof(TRACEFILE) or B = DBLK;
    if B = DBLK then
      B := GetNextBlock(TRACEFILE);
      CONTINUE := true;
      while CONTINUE do
        if IsACall(B) then
          Push(PROCSTACK,B);
          VAR := GetFormalName(VAR,B);
          DEFSET := SetOfBlocks(VAR);
        elseif IsAReturn(B) then
          VAR := GetActualName(VAR,Pop(PROCSTACK));
          DEFSET := SetOfBlocks(VAR);
        elseif B = UBLK1 then
          if TY = c-use then ACCEPT := true;
          elseif TY = p-use then
            B := GetNextBlock(TRACEFILE);
            if B = UBLK2 then ACCEPT := true;
          endif;
        elseif B ∈ DEFSET then CONTINUE := false;
        endif;
        B := GetNextBlock(TRACEFILE);
      endwhile;
    endif;
  endwhile;
  Accept := ACCEPT;
end

```

Figure 4: Algorithm Accept

local information throughout the graph is accomplished in step three. If no recursion is present, only one iteration of the algorithm is required to compute the global information. This requires a reverse topological ordering of the nodes in each procedure along with an invocation ordering on the procedures themselves. Thus, for non-

recursive procedures, the algorithm performs in linear time.

For programs with recursive procedures, steps two and three depend on the number of procedures. For step two, in the worst case, the processing may visit each node p times where p is the number of procedures in the program and thus, the time is $O(pn)$. In step three, p iterations may be required to obtain the IN and OUT sets. Thus, the algorithm is $O(pn)$ when recursion is present.

4.2. INTERPROCEDURAL DATA FLOW TESTING

After the interprocedural data flow information is computed and the required definition-use pairs are determined, the tester guides the selection and execution of the module with test cases as input. This consists of choosing the required definition-use pairs according to the desired testing criterion and processing the test cases until the required pairs are satisfied. The required definition-use pairs depend on the desired testing criterion. For example, if the criterion is 'all-p-uses/some-c-uses', the tester runs the acceptor with all definition-p-use pairs. If for a particular definition, no p-use exists, then some definition-c-use pairs must be accepted.

Processing a test case consists of (1) executing the module with the test data as input to get the test path and (2) running the test case acceptor with the test path and a definition-use pair as input. Our tester instruments the module at the intermediate code level. Intermediate code statements are inserted that output to a file the number of each basic block that is traversed during module execution. However, the execution path must also contain information that signals procedure calls and returns so that the renaming of formal and actual reference parameters can be handled. We accomplish this by instrumenting the intermediate code to indicate procedure calls and returns. The algorithm for the acceptor is given in Figure 4. It inputs the file containing the block numbers of the execution path (TRACEFILE) along with the information about the definition-use pair that is to be tested (VAR, DBLK, UBLK1, UBLK2, TY). VAR represents the variable being considered and DBLK represents the block number containing the desired definition. TY is either c-use or p-use. If TY is c-use, then UBLK1 contains the block number of the desired use and UBLK2 is not used; if TY is p-use, then (UBLK1, UBLK2) represents the desired edge. Since we also have access to the data flow information that was computed in the first phase, it is used to determine the set of blocks containing definitions, DEFSET, of the actual parameter at returns from procedures and the formal parameter at the entries to procedures. The entry into a procedure causes the DEFSET to be changed to reflect the renaming of the formal parameter that is bound to the definition being tested. The exit from a procedure causes the DEFSET to revert back to its value in

the calling procedure.

To illustrate, consider the test case with input 4,8,9,2 where the definition-use pair being processed is the definition of K in B12 of procedure PairMax and the use is the edge (B10,B11). This definition of K reaches the use of I on edge (B10,B11) since it reaches over the return to calling procedure GetMax, over the return to GetMax through the recursive call, over the call to GetMax again, and finally over the call to PairMax where the variable is bound to I. The steps involved in processing the test path are shown below. The DEFSET is initialized to the set of definitions of the variable corresponding to the definition that is being tested. At each call to and return from a procedure, the DEFSET is changed to reflect the new name of the actual or formal parameter. Since there are no interprocedural definitions of the corresponding variable MX in GetMax, the DEFSET is empty while processing the blocks in that procedure. The partial test paths are indented to show the procedure nesting along the execution path.

execution subpath	action taken	DEFSET
1,2,4,5,GetMax	Push(5)	{B11,B12}
1,2,3,PairMax	Push(3)	{B11,B12}
9,10,12	DBLK found	{B11,B12}
13,Return	Pop(3); reset VAR; assign DEFSET	ϕ
8,Return	Pop(5); reset VAR; assign DEFSET	ϕ
6,GetMax	Push(6); reset VAR; assign DEFSET	ϕ
1,2,3,PairMax	Push(3); reset VAR; assign DEFSET	{B11,B12}
9,10,11	use edge found \Rightarrow accept test case	{B11,B12}

5. CONCLUSIONS

In this paper, we have extended data flow testing to include the testing of definitions that reach, and uses that can be reached, across procedure calls and returns. To do so, required the development of an interprocedural data flow analysis technique that computes the locations of interprocedural reaching definitions and use pairings and the development of a testing methodology that associates actual and formal parameters in calls and returns. The benefit of the resulting testing system is that data flow testing can be uniformly applied to individual procedures for the integration of procedures in a module and to the interfaces of procedures.

The interprocedural data flow technique has been implemented and tested on a Sun 3/50 Workstation. Additionally, the interprocedural definition-use information has been used in the the implementation of the interprocedural tester that utilizes the data flow information.

In order to avoid retesting of all interprocedural data dependencies in a module when a modification is made to one procedure, algorithms have been developed that efficiently retest the modified procedure and its interactions by first identifying and then retesting only those interactions of the module affected by the change. The need for complete retesting is eliminated.[12, 13] The nodes and edges of the portion of the IFG associated with the modified procedure are updated to reflect the changes. The control flow graph of no procedure other than the modified procedure needs to be recomputed and the structure of the IFG remains unchanged for these unmodified procedures. This algorithm determines the affected definition use pairings which are then reported to the user for retesting.

The interprocedural data flow tester described in this paper illustrates our technique for testing with criteria other than 'all-du-paths'. Although requiring that the 'all-du-paths' criterion be satisfied may be prohibitive in most cases because of the numerous subpaths that would exist interprocedurally, some applications justify this level of testing. In this case, additional information about the subpaths involving the interprocedural definitions and uses is obtained during intraprocedural analysis and used to construct the interprocedural du-paths that are required for the testing.

Future work includes experimentation with the data flow tester to determine the cost of applying interprocedural data flow testing, especially when changes are made within a module. In addition, we also intend to apply the same principle of summarizing information to determine the problems of developing an intermodular data flow testing technique. Also, although aliases can be incorporated into the original program summary graph, we have yet to incorporate it into our testing system.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman, in *Compilers, Principles, Techniques, and Tools*, Addison-Wesley Publishing Company, Massachusetts, 1986.
2. F. E. Allen, "Interprocedural data flow analysis," in *IFIP Information Processing 74*, North-Holland Publishing Company, 1974.

3. J. P. Banning, "An efficient way to find the side effects of procedure calls and aliases of variables," *Sixth Annual ACM Symposium on Principles of Programming Languages*, pp. 29-41, January 1979.
4. J. M. Barth, "A practical interprocedural data flow analysis algorithm," *CACM*, vol. 21, no. 9, pp. 724-736, September 1978.
5. D. Callahan, "The program summary graph and flow-sensitive interprocedural data flow analysis," *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pp. 47-56, Atlanta, GA, June 1988.
6. M. D. Carroll and B. G. Ryder, "An incremental algorithm for software analysis," *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices*, vol. 22, no. 1, January 1987.
7. M. D. Carroll and B. G. Ryder, "Incremental data flow analysis via dominator and attribute updates," *Proceedings of the Fifteenth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, San Diego, CA, January 1988.
8. 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, London, UK, August 1985.
9. K. Cooper and K. Kennedy, "Interprocedural side-effect analysis in linear time," *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pp. 57-66, Atlanta, GA, June, 1988.
10. P. G. Frankl, S. N. Weiss, and E. J. Weyuker, "ASSET: A system to select and evaluate tests," *Proceedings of the IEEE Conference on Software Tools*, New York, April 1985.
11. P. G. Frankl and E. J. Weyuker, "An applicable family of data flow testing criteria," *IEEE Transactions on Software Engineering*, vol. 14, no. 10, pp. 1483-1498, October 1988.
12. M. J. Harrold, "An approach to incremental testing," Technical Report 89-1 Department of Computer Science, University of Pittsburgh, January 1989.
13. M. J. Harrold and M. L. Soffa, "An incremental data flow testing tool," *Proceeding of the Sixth International Conference on Testing Computer Software*, Washington, DC, May 1989.
14. B. Korel and J. Laski, "A tool for data flow oriented program testing," *ACM Softfair Proceedings*, pp. 35-37, December 1985.
15. J. W. Laski and B. Korel, "A data flow oriented program testing strategy," *IEEE Transactions on Software Engineering*, vol. SE-9, no. 3, pp. 347-354, May 1983.
16. D. B. Lomet, "Data flow analysis in the presence of procedure calls," *IBM Journal of Research and Development*, vol. 21, no. 6, pp. 559-571, November 1977.
17. E. W. Myers, "A precise inter-procedural data flow algorithm," *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*, pp. 219-230, Williamsburg, VA, January 1981.
18. S. C. Ntafos, "An evaluation of required element testing strategies," *Proceedings 7th International Conference on Software Engineering*, pp. 250-256, Orlando, Florida, March 1984.
19. S. Rapps and E. J. Weyuker, "Selecting software test data using data flow information," *IEEE Transactions of Software Engineering*, vol. SE-11, no. 4, pp. 367-375, April 1985.