Efficient Computation of Interprocedural Definition-Use Chains

MARY JEAN HARROLD
Clemson University

and

MARY LOU SOFFA
University of Pittsburgh

The dependencies that exist among definitions and uses of variables in a program are required by many language-processing tools. This paper considers the computation of definition-use and use-definition chains that extend across procedure boundaries at call and return sites. Intraprocedural definition and use information is abstracted for each procedure and is used to construct an interprocedural flow graph. This intraprocedural data-flow information is then propagated throughout the program via the interprocedural flow graph to obtain sets of reaching definitions and/or reachable uses for each interprocedural control point, including procedure entry, exit, call, and return. Interprocedural definition-use and/or use-definition chains are computed from this reaching information. The technique handles the interprocedural effects of the data flow caused by both reference parameters and global variables, while preserving the calling context of called procedures. Additionally, recursion, aliasing, and separate compilation are handled. The technique has been implemented using a Sun-4 Workstation and incorporated into an interprocedural data-flow tester. Results from experiments indicate the practicality of the technique, both in terms of the size of the interprocedural flow graph and the size of the data-flow sets.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.6 [**Software Engineering**]: Programming Environments; D.3.4 [**Programming Languages**]: Processors—*compilers; optimization*

General Terms: Algorithms, Design

Additional Key Words and Phrases: Data-flow testing, interprocedural data-flow analysis, interprocedural definition-use chains, interprocedural reachable uses, interprocedural reaching definitions

This work was partially supported by the National Science Foundation under Grant CCR-9109531 to Clemson University and Grant CCR-9109089 to the University of Pittsburgh. An earlier version of this paper appeared in abridged form in *Proceedings of the International Conference on Computer Languages 1990*. © Copyright 1987 and 1989, Free Software Foundation, Inc., 675 Mass. Avenue, Cambridge, MA 02139.
Authors' addresses: M. J. Harrold, Department of Computer Science, Clemson University, Clemson, SC 29634-1906; M. L. Soffa, Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 15260.

1. INTRODUCTION

An important component of compilation is data-flow analysis that computes information about the potential flow of data throughout a program. Although data-flow analysis is traditionally used in the last two phases of a compiler, namely, code optimization and code generation, it has also become an integral part of other language-processing tools, such as editors, parallelizers, debuggers, anomaly checkers, and testers. *Intra*procedural data-flow analysis considers the flow of data within a procedure, while assuming some approximation about definitions and about uses of reference parameters and global variables at call sites. *Inter*procedural data-flow analysis computes information about the flow of data across procedure boundaries caused by reference parameters and global variables.

Although interprocedural data-flow analysis techniques do exist [Banning 1979; Barth 1978; Burke 1990; Callahan 1988; Cooper et al. 1986a; Horwitz et al. 1990; Meyers 1981], they are insufficient for computing the locations of definition-use pairs that extend across procedure boundaries. One such application that requires this information is data-flow testing [Frankl and Weyuker 1988; Harrold and Soffa 1991; Laski and Korel 1983; Ntafos 1984] in which test-data adequacy criteria are used to select particular definition-use pairs as the test-case requirements for the program. For example, the "all-uses" criterion requires that each definition be tested on some path to each of its uses. A technique that performs data-flow testing for a program with many procedures requires both intraprocedural and interprocedural definition-use pairs. After the definition-use pairs are identified, test cases are generated that satisfy the requirements when used in the program's execution. A test case is said to *satisfy* a definition-use pair if the execution of the program with the test case traverses a path from the definition to the use without any intervening redefinition of the variable. To determine particular definition-use pairs, data-flow testing requires either *definition-use chains* or *use-definition chains*.

Analysis techniques for computing definition-use (use-definition) chains for individual procedures [Aho et al. 1986, pp. 632–633] are well known and have been used in various tools, including data-flow testers [Frankl and Weyuker 1985; Harrold and Soffa 1989; Korel and Laski 1985; Taha et al. 1989]. To provide the necessary data-flow information to test interprocedural definitions and uses, *interprocedural definition-use chains* are required. An interprocedural definition-use chain for a definition in some procedure **P** consists of the locations of all uses of the definition, taking into account uses in procedures that are reachable from **P** along both call and return sequences. The computation of interprocedural definition-use chains requires tracking the uses of global variables and formal and actual reference parameters that can be reached across call and return sites (i.e., *interprocedural reachable uses*). *Interprocedural use-definition chains* are defined analogously in terms of *interprocedural reaching definitions*.

In this paper we present an efficient interprocedural analysis algorithm that computes interprocedural definition-use and use-definition chains. Our approach is modeled after the iterative computation of intraprocedural data-flow chains. We first analyze individual procedures in a program in any order to abstract intraprocedural information, which is used to construct an *interprocedural flow graph* (*IFG*). For interprocedural definition-use chains, we propagate intraprocedural use information throughout the program via the IFG to obtain the interprocedural reachable uses for procedure control points in the program, while taking into account the calling context of called procedures. We compute the interprocedural definition-use chains using local definition information at each node in the IFG along with the propagated reachable use information. Interprocedural use-definitions chains are computed similarly, except that interprocedural reaching definitions are propagated throughout the program. To be applicable to large programs [Cooper et al. 1986b], the data-flow analysis technique supports separate compilation, since procedures are analyzed in isolation of one another. The interprocedural data-flow analysis technique computes interprocedural definition-use (use-definition) chains for recursive procedures, making its use applicable to a wider range of programs. The technique can also incorporate previously determined alias information to compute safe interprocedural definition-use (use-definition) chains. We have implemented a prototype of our system in C and have performed experiments to determine the size of both the IFG and the data-flow sets. Our experiments show that, in practice, the IFG is linear in the size of the program and that the size of the data-flow sets is small compared to the number of statements in the program.

In the remainder of this paper, we present our algorithms for computing interprocedural definition-use and use-definition chains. The problems of gathering interprocedural definition and use information are discussed in Section 2. Next, we focus on our technique for computing definition-use and use-definition chains for reference parameters and global variables in an alias-free environment, since the propagation algorithms are the same when aliasing is present. Section 3 discusses our interprocedural program representation. Section 4 presents the details of our algorithm for computing definition-use chains for reference parameters and then for global variables. We first detail our algorithm for computing interprocedural definition-use chains and then present the analogous technique for computing interprocedural use-definition chains. In Section 5 we discuss our technique for incorporating alias information into our program representation and computing safe interprocedural definition-use (use-definition) chains in the presence of aliasing. Our experimental results are given in Section 6, related work is discussed in Section 7, and concluding remarks are given in Section 8.

2. COMPUTING INTERPROCEDURAL DEFINITION-USE CHAINS

The problems involved in computing interprocedural data-flow chains include accommodating separate compilation of procedures, handling programs with recursive procedures, ensuring that data-flow chains reflect all, and only, definition and use pairings for possible control paths in the program, and providing safe information in the presence of aliases. The program in Figure 1 illustrates some of these difficulties. To assist the reader in tracking

```
program Main
   read(X);
   P1(X);
   if X>20 then X:=X+6;
   P2(X);
   write(X);
end Main;
```

```
procedure P1(Y)
   if Y<10 then Y := Y + 4;
   P2(Y);
   if Y<25 then Y := Y + 5;
end P1;
```

```
procedure P2(Z)
   if Z > 10 then Z := Z + 2;
end P2;
```

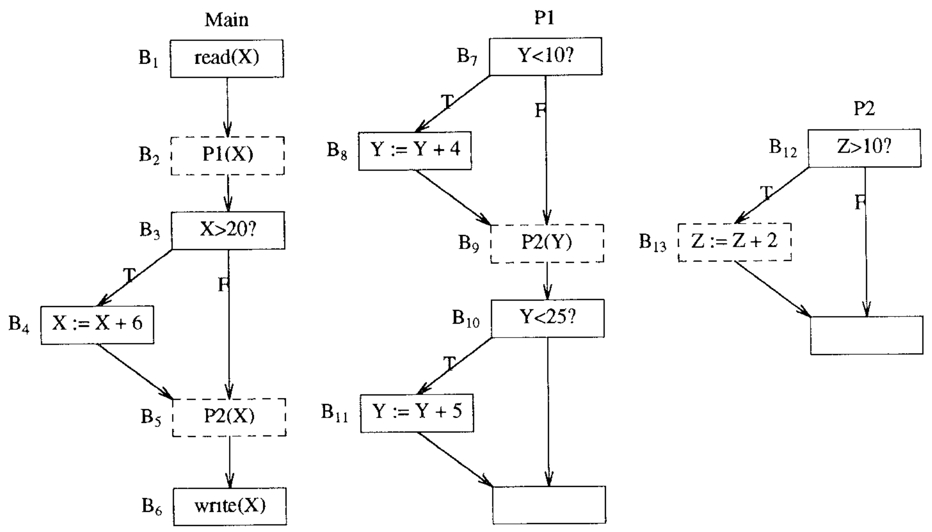


Fig 1. Example program Main and its control flow graphs.

interprocedural definition-use dependencies, each procedure's *control flow graph* [Aho et al. 1986, p. 532] is depicted beneath its code. A node in a control flow graph represents a basic block in the program, consisting of statements that execute sequentially, and an edge represents the flow of control between basic blocks. To simplify the interprocedural analysis, we represent a call site by a single basic block, shown as a dashed box in Figure 1.

In Figure 1 information about the interprocedural data flow in procedures P1 and P2 is required to determine that the definition of variable X in B_1 may be preserved [Lomet 1977] over the call to procedure P1 in B_2 and consequently reaches the use of X in B_3. The analysis for providing this information must also consider the call to procedure P2 from procedure P1. In this case, interprocedural information is being used to determine the local effects of the procedure call on the *calling* procedure. Gathering this information requires that either (1) the information about called procedures be incorporated at call sites during the analysis of the calling procedure or (2) an estimate of the information about called procedures be used during initial analysis of the calling procedure and that this information be updated when more accurate data-flow information is determined. The problem with the first method is that, if procedures are processed in any order or are recursive, incomplete information may be available about called procedures at call sites. Thus, in this paper we use the second method. We process each procedure individually to abstract the intraprocedural information, and we use an estimate of the definition and use information at call sites. This initial estimate is updated by propagating information about other procedures to obtain the interprocedural data-flow information.

Tracking the definition of X in B_1 in Figure 1 over procedure calls and returns is required to determine that this definition of X reaches the uses of Y in B_7, B_8, B_{10}, and B_{11} of P1 and the uses of Z in B_{12} and B_{13} in P2. In this case, interprocedural information is being used to determine the interprocedural definition and use dependencies in the *called* procedures. Tracking requires that data-flow information be propagated across procedure calls and returns throughout the program.

Preserving the *calling context* of called procedures is important during the computation of data-flow chains. To preserve the calling context, only those paths through the program that agree with the call sequence for some possible control path should be traversed when tracking data-flow pairs over returns from procedures. Thus, either the call sequence must be "remembered" or the technique for propagation must account for actual call sequences. Consider the definition of X in B_4 that reaches the call to procedure P2 in B_5. Since there is a path through procedure P2 on which Z is not defined, the definition of X in B_4 can reach the end of procedure P2. However, since there are two calls to P2, there are two return paths from procedure P2: one that returns directly to Main and the other that returns indirectly through P1. Ignoring the call sequence suggests that the definition in B_4 has uses in B_3, B_4, and B_6. However, closer inspection of control paths through the program reveals that this definition in B_4 reaches the end of P2, and subsequently back into Main, only when it is called directly from Main. Thus, this definition can only reach the use of X in B_6. The calling context must be considered in order to obtain more precise definition-use chains.

The program in Figure 2 illustrates one difficulty in computing precise interprocedural data-flow chains when aliasing is present. An alias is introduced at the call site in $s2$ since actual parameter X is passed in two locations in the parameter list. Thus, on this call to P1, formal parameters Y and Z are aliases of each other. Then, in $s5$, this alias is propagated to P2 since Y and Z are passed as parameters, causing A and B to be aliased on this call to P2. Without considering the effects of aliasing, the only definition in P2 that reaches the use of Y in $s6$ is the definition of A in $s8$. More precise analysis reveals that, when Y and Z are aliased, the definition of B in $s9$ also reaches the use of Y in $s6$. To provide safe data-flow chains, the interprocedural analysis algorithms must consider the effects of aliasing.

3. THE IFG

We represent a program, composed of a number of procedures, by an *interprocedural flow graph* (*IFG*), which is based on the program summary graph [Callahan 1988]. Like the program summary graph, an IFG contains one

```
program Main             procedure P1(Y,Z)        procedure P2(A,B)
  s1: read(X);             s4: if Y<10              s7: if B > 0
  s2: P1(X, X);            s5:    then P2(Y,Z);     s8:    then A := A + B
  s3: write(X);            s6: write(Y);            s9:    else B := A - B;
end Main;                end P1;                  end P2;
```

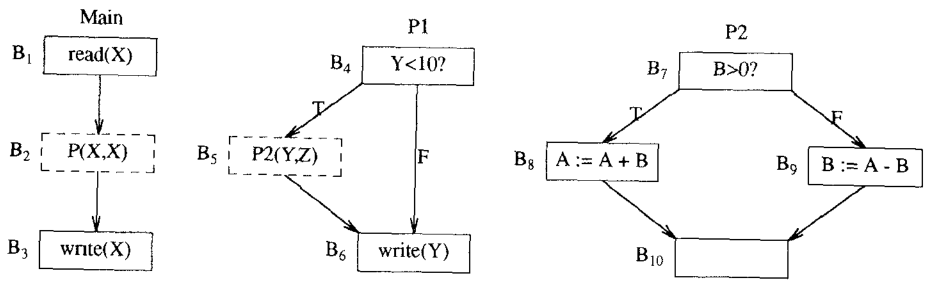


Fig. 2. Example program and its control flow graphs, illustrating one of the problems in computing data-flow chains in the presence of aliasing.

subgraph for each procedure. Each IFG subgraph represents local information about the procedure that is abstracted from intraprocedural analysis and includes information about formal parameters in the procedure and actual parameters at call sites in the procedure. Information about the interaction among procedures is used to connect the subgraphs to get the IFG for the program. The IFG is constructed by considering binding information about formal and actual parameters and control information that can be determined about a procedure and abstracted to its call sites.

The four types of nodes in a program summary graph, *entry*, *exit*, *call*, and *return*, are also nodes in an IFG and correspond to control points before and after regions of code in the associated procedure. An entry node represents the point prior to entry into the procedure, and an exit node represents the point after the end of the procedure. A call node represents the point prior to the procedure call whereas a return node represents the point after the return from the procedure call. A call node and a return node are created for each actual parameter at a call site, and an entry node and an exit node are created for each formal parameter in a procedure. Intraprocedural, or local, information is computed about definitions of formal and actual parameters that reach, and uses of formal and actual parameters that can be reached from, these control points and is attached to the appropriate nodes in the subgraph. At entry and exit points of a procedure, local definition and use information is abstracted for formal parameters, while at call and return sites, local definition and use information is abstracted about actual parameters involved in the call.

Reaching edges from both entry and return nodes to call and exit nodes abstract the control information from the procedure by indicating that a definition that reaches the source of the edge also reaches the sink of the edge. For example, a reaching edge from an entry node to a call node indicates that a definition of the formal parameter that reaches the beginning of the procedure also reaches the call site where it is used as an actual parameter. A reaching edge is strictly intraprocedural since it is computed without incorporating the control structure of the called procedure at a call site. The computation treats each call node as a use of its actual parameter and each return node as a definition, with no definitions between the call and return nodes.

While nodes and reaching edges in the IFG subgraphs are identical to those in the program summary graph, each IFG contains interreaching edges and additional definition and use information about each procedure. We compute sets of definitions of formal and actual parameters that reach interprocedural control points and sets of uses of formal and actual parameters that can be reached from interprocedural control points, and attach them to nodes in the IFG subgraphs. Our algorithm propagates these sets throughout the program using the IFG to obtain interprocedural reaching definitions and reachable uses. Computing the sets of definitions of formal and actual parameters in a procedure that reach the ends of the appropriate regions of code represented by nodes in the IFG, designated the DEF sets for the nodes, is similar to computing *generated definition* sets for basic blocks in intraprocedural data-flow analysis. The definitions of an actual parameter that reach a call site constitute the DEF set for the call node associated with that actual parameter at the call site, while definitions of a formal parameter that reach the end of the procedure become the DEF set for the exit node associated with that formal parameter at the end of the procedure. A similar situation exists for reachable uses. Information about uses of formal parameters that can be reached from the beginning of a procedure and of actual parameters that can be reached from the return of a procedure is gathered at entry and exit nodes, respectively. Computing the sets of uses, UPEXP, that can be reached from the beginning of the regions of code represented by entry and return nodes in the IFG is similar to computing the *upward exposed* sets of uses for basic blocks in intraprocedural data-flow analysis. The UPEXP set at an entry node for a formal parameter is the set of uses of that formal parameter that can be reached from the start of the procedure. Likewise, the UPEXP set at a return node for an actual parameter is the set of uses of that actual parameter that can be reached from the return from a procedure call. The DEF sets for entry and return nodes and the UPEXP sets for call and exit nodes have no meaning intraprocedurally and are given a null value to facilitate propagating the data-flow information. For example, any definitions of actual parameters that can reach the beginning of the procedure must be computed from interprocedural analysis and, thus, are not known during the construction of the subgraphs.

In an IFG, procedure entry and procedure exit are denoted by entry_f^P and exit_f^P nodes, respectively, and both nodes are created for every formal parameter **f** of every procedure **P**. Procedure invocation and procedure return are represented by $\text{call}_x^{P \to Q}$ and $\text{return}_x^{P \to Q}$ nodes, respectively, and both nodes are created for every actual parameter **x** of every call from procedure **P** to procedure **Q**. Since each node represents a single variable, the DEF and UPEXP sets at a node correspond to a single formal or actual parameter.

```
algorithm ComputeChains(G, P)
    /* restricted to definition-use chains for reference parameters */
input     G: an IFG
          P: a collection of procedures
declare   G_i: an IFG subgraph
          NODESET: set of nodes to process
          EDGESET: set of edges to process
          DUC: array of definition-use chains
begin
/* Step 1: subgraph construction for each procedure */
    for each $P_i \in P$ do                                /* process each procedure */
        for each formal f do create $entry_f^{P_i}$, $exit_f^{P_i}$
        for each actual x at P->Q do create $call_x^{P_i->Q}$, $return_x^{P_i->Q}$
        Perform intraprocedural data flow analysis on $P_i$
        Using the intraprocedural data flow information
            Create reaching edges for $P_i$
            Extract DEF[k] and UPEXP[k], $k \in G_{P_i}$
    endfor
/* Step 2: construction of the IFG */
    Create the binding edges among the $G_{P_i}$
    Determine may-be-preserved information for each entry node
    Create interreaching edges
/* Step 3: IFG propagation to obtain global information */
    for each node n in G do                                /* initialization */
        $IN_{Use}[n]$ = UPEXP[n]
        $OUT_{Use}[n] = \phi$
    endfor
    NODESET := {entry, call, return nodes in G}            /* phase 1 */
    EDGESET := {all edges in G}
    Propagate(NODESET, EDGESET)
    NODESET := {all nodes in G}                            /* phase 2 */
    EDGESET := {return binding, reaching, interreaching edges in G}
    Propagate(NODESET, EDGESET)
/* Step 4: interprocedural definition-use chains computation */
    for each $P_i \in P$ do
        for each interprocedural definition d in $P_i$ do
            DUC[d] := $\phi$
            if $d \in DEF[call_x^{P_i->Q}]$ then DUC[d]:=DUC[d] $\cup$ $OUT_{Use}[call_x^{P_i->Q}]$
            if $d \in DEF[exit_f^P]$ then DUC[d] := DUC[d] $\cup$ $OUT_{Use}[exit_f^P]$
        endfor
    endfor
end ComputeChains
```

Fig. 3. Algorithm for computing interprocedural definition-use chains for reference parameters.

The interaction among the procedures is used to connect the IFG subgraphs. Binding edges in the IFG from call nodes to entry nodes and from exit nodes to return nodes correspond to the bindings of the formal and actual parameters. *Interreaching* edges from call nodes to return nodes abstract the control information about the called procedures at call sites. This edge indicates that a definition that reaches the procedure call may be preserved after the return from the procedure. The interreaching edges allow the calling context of the called procedures to be preserved during propagation.

4. INTEREPROCEDURAL DEFINITION-USE CHAINS

We now present our algorithm for constructing the IFG and for computing interprocedural definition-use chains for a program. We model our algorithm after the iterative data-flow analysis technique used at the intraprocedural level: (1) Local information about definitions and uses is gathered at points in the program before and after regions of code that correspond to nodes in the graph representation of the program, (2) iterative techniques are used to solve data-flow equations for reaching definitions and reachable uses by propagating the local information throughout the graph, and (3) data-flow chains are computed by associating the local information gathered in (1) with the propagated information determined in (2). At the *intra*procedural level, the regions of code are basic blocks, the definition and use information is gathered at points before and after these blocks, and the graph is a control flow graph. At the *inter*procedural level, the regions represent parts of the program that are of interest interprocedurally (e.g., the regions between procedure calls), the definition and use information is gathered at points before and after these regions (i.e., the procedure control points), and the graph is an interprocedural flow graph. We present our algorithm for computing interprocedural data-flow chains by first detailing the computation of interprocedural definition-use chains for reference parameters. Our algorithm *ComputeChains*, which computes the four steps for the interprocedural definition-use chains for reference parameters, is given in Figure 3. Analogous computation of interprocedural use-definition chains is discussed in Section 4.5, and a technique that handles global variables is discussed in Section 4.6.

4.1 Step 1: Construct the IFG Subgraphs

In the first step, we process each procedure **P** once and construct its IFG subgraph. Intraprocedural data-flow analysis is performed on **P**, and local information is abstracted to the IFG subgraphs. The local information consists of the DEF and UPEXP sets for regions of code in **P**, along with the control information about **P**. The DEF and UPEXP sets are attached to nodes in the IFG subgraph, and the control information is used to construct the reaching edges. To compute the DEF and UPEXP sets, basic blocks are added at the beginning (i.e., the *initial* block) and at the end (i.e., the *final* block) of the control flow graph for **P**. During the intraprocedural data-flow analysis of **P**, reaching information is gathered at these basic blocks.

The DEF and UPEXP sets are defined as follows, where **x** represents an actual parameter and **f** represents a formal parameter:

$$\mathrm{DEF}[n] = \begin{cases} \text{definitions of x in P that reach P} \rightarrow \text{Q} & \text{if n is } \mathrm{call}_x^{P \rightarrow Q}, \\ \text{definitions of f in P that reach the end of P} & \text{if n is } \mathrm{exit}_f^{P}, \\ \phi & \text{otherwise;} \end{cases}$$

$$\mathrm{UPEXP}[n] = \begin{cases} \text{uses of f in P reachable from the beginning of P} & \text{if n is } \mathrm{entry}_f^{P}, \\ \text{uses of x in P reachable from the return from P} \rightarrow \text{Q} & \text{if n is } \mathrm{return}_x^{P \rightarrow Q}, \\ \phi & \text{otherwise.} \end{cases}$$

To compute the control information for **P**, dummy definitions of the formal parameters are added to the initial block, and dummy uses of the formal parameters are added to the final block. Dummy definitions and uses of actual parameters are also added to the basic blocks, containing the call sites. These dummy definitions and uses facilitate the gathering of the reaching information during the intraprocedural analysis of **P**. For example, if a dummy definition of formal parameter **f** in the initial block reaches the beginning of the final block, a reaching edge is constructed that connects the entry node for **f** with the exit node for **f** in the IFG subgraph for **P**.

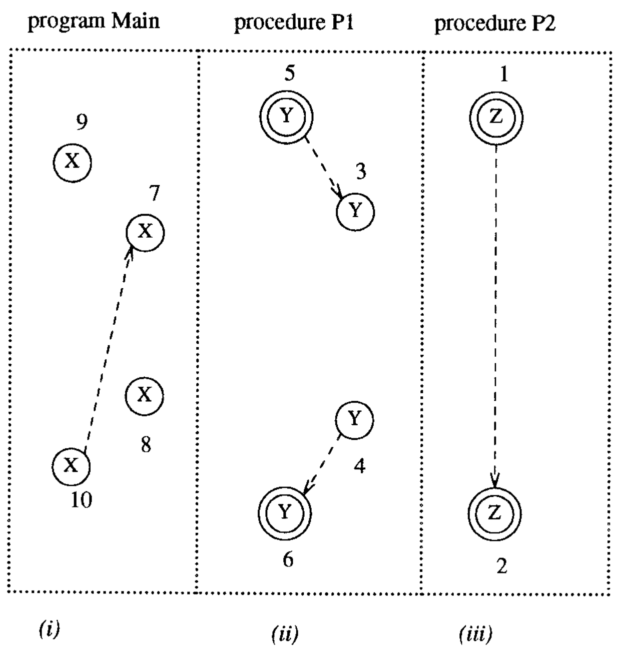
Consider the program given in Figure 4 which differs from the program in Figure 1 in that the statements are numbered and the points corresponding to nodes in the IFG are marked. Statements are numbered consecutively throughout the program, and program points corresponding to nodes in the graph are indicated. In the IFG subgraphs, circles represent call and return nodes, doubled circles represent entry and exit nodes, and dashed lines represent reaching edges. The subgraphs (*i*), (*ii*), and (*iii*) are constructed for program Main and procedures P1 and P2, respectively. Consider subgraph (*ii*), representing procedure P1, where nodes 3, 4, 5, and 6 are created for the call to P2, the return from P2, the entry to P1, and the exit from P1, respectively. Since a definition of an actual parameter associated with formal parameter Y that reaches the beginning of procedure P1 also reaches the call to procedure P2 where it is used as a parameter, reaching edge (5, 3) is created. For node 3, inspection of the data flow in P1 reveals that the definition of Y in *s*6 is the only local definition that reaches the call site, and thus, DEF[3] is {Y in *s*6}. For simplicity, we use only {*s*6} to denote this set unambiguously.

Next, consider node 6. Since it is assumed that, during construction of the IFG subgraphs, definitions are not preserved over procedure calls, the only definition reaching the end of procedure P1 that can be determined intraprocedurally is the definition of Y in *s*8. Note that, since the associated formal parameter Z may be preserved over a call to procedure P2, the definition in

```
program Main
  s1: read(X);
  s2: P1(X);  {nodes 9, 10}
  s3: if X>20 then X:=X+6;
  s4: P2(X);  {nodes 7, 8}
  s5: write(X);
end Main;
```

```
procedure P1(Y)  {node 5}
  s6: if Y<10 then Y := Y + 4;
  s7: P2(Y);  {nodes 3, 4}
  s8: if Y<25 then Y := Y + 5;
end P1;  {node 6}
```

```
procedure P2(Z)  {node 1}
  s9: if Z > 10 then Z := Z + 2;
end P2;  {node 2}
```



node	DEF	UPEXP
1	ϕ	{s9}
2	{s9}	ϕ
3	{s6}	ϕ
4	ϕ	{s8}
5	ϕ	{s6}
6	{s8}	ϕ
7	{s3}	ϕ
8	ϕ	{s5}
9	{s1}	ϕ
10	ϕ	{s3}

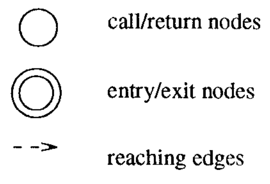
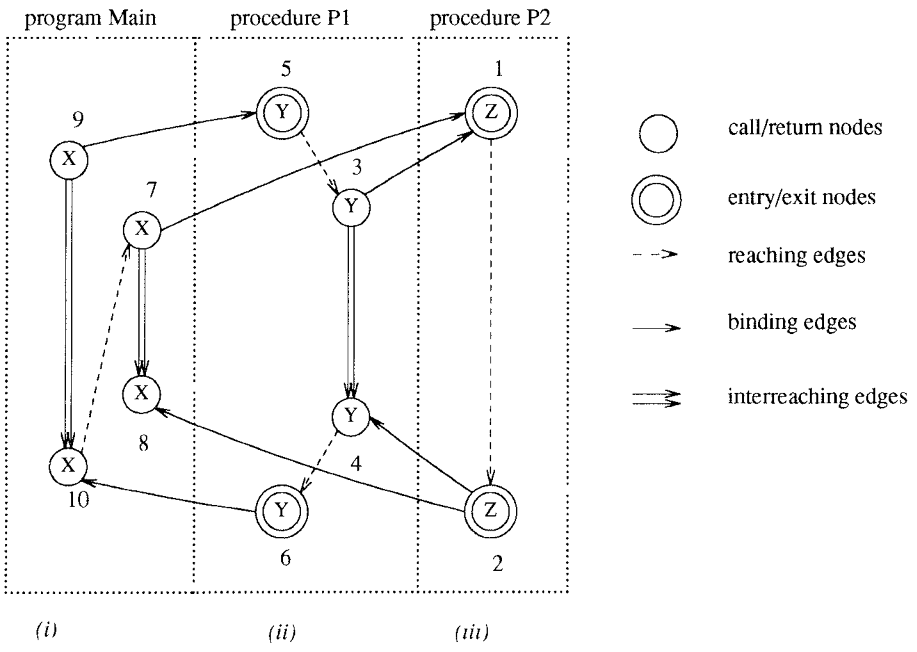


Fig. 4. Example program Main and its interprocedural flow graph's subgraphs (*i*), (*ii*), and (*iii*). The DEF and UPEXP sets attached to nodes in the subgraphs are also shown.

$s6$ also reaches the end of procedure P1. This information cannot be computed intraprocedurally and, thus, is not part of the DEF set, but is obtained during the propagation step. The use of Y in $s6$ is reachable from the beginning of the procedure, and thus, $s6$ is in UPEXP[5]. The use of Y in $s8$ is reachable from the return from procedure P2, and thus, Y in $s8$ is in UPEXP[4]. Finally, the reaching edge (4, 6) indicates that there is a path from the return from procedure P2 in $s7$ to the end of procedure P1, along which Y is not defined.

4.2 Step 2: Construct the IFG

After all procedures have been processed, the appropriate binding edges among actual and formal parameters in the procedures are added in the construction of the IFG. The last step is to add the interreaching edges by processing the partially constructed IFG using an iterative algorithm [Callahan 1988] to determine whether, for each $\text{entry}_{\text{f}}^{\text{P}}$, **f** may be preserved.



propagated data flow information				
node	IN_{Use}	OUT_{Use}	IN_{Def}	OUT_{Def}
1	*{s3,s5,s8,s9}*	*{s3,s5,s8,s9}*	*{s1,s3,s6,s8,s9}*	*{s1,s3,s6,s8,s9}*
2	*{s3,s5,s8,s9}*	*{s3,s5,s8,s9}*	*{s1,s3,s6,s8,s9}*	*{s1,s3,s6,s8,s9}*
3	*{s3,s5,s8,s9}*	*{s3,s5,s8,s9}*	*{s1}*	*{s1,s6}*
4	*{s3,s5,s8,s9}*	*{s3,s5,s9}*	*{s1,s6,s9}*	*{s1,s6,s9}*
5	*{s3,s5,s6,s8,s9}*	*{s3,s5,s8,s9}*	*{s1}*	*{s1}*
6	*{s3,s5,s9}*	*{s3,s5,s9}*	*{s1,s6,s9}*	*{s1,s6,s8,s9}*
7	*{s5,s9}*	*{s5,s9}*	*{s1.s6,s8,s9}*	*{s1,s3,s6,s8,s9}*
8	*{s5}*	ϕ	*{s1,s3,s6,s8,s9}*	*{s1,s3,s6,s8,s9}*
9	*{s3,s5,s6,s8,s9}*	*{s3,s5,s6,s8,s9}*	ϕ	*{s1}*
10	*{s3,s5,s9}*	*{s5,s9}*	*{s1,s6,s8,s9}*	*{s1,s6,s8,s9}*

Fig 5. Interprocedural flow graph for program Main in Figure 2.

The iterative algorithm to compute the may-be-preserved information uses a worklist to keep track of the nodes that are to be processed. Initially, the may-be-preserved value for each of the exit nodes is initialized to *true*, while at all other nodes, it is initialized to *false*. The algorithm propagates the may-be-preserved information backward throughout the partially constructed IFG as far as it can reach; the may-be-preserved values for those nodes that are reachable from the exit nodes are changed. Interreaching edges are created for each call-return pair whose associated entry node's may-be-preserved value is *true*. This step completes the construction of the IFG.

To illustrate the results of Step 2, consider Figure 5, where the subgraphs of Figure 4 are connected. Step 2 connects subgraphs (*i*), (*ii*), and (*iii*) by adding the binding edges to connect formal and actual parameters. Edges (3, 1) and (2, 4) indicate the binding of actual parameter Y in procedure P1 to formal parameter Z in procedure P2. Additional binding edges added during Step 2 are edges (9, 5) and (6, 10), representing the call to procedure P1 in Main, and edges (7, 1) and (2, 8), representing the call to procedure P2 in Main. An iterative algorithm [Callahan 1988] is used to compute the may-be-preserved information for formal parameters Z and Y, represented by nodes 1 and 5, respectively, in the graph. Since the results of this analysis indicate that information reaching the entry nodes may also reach the exit nodes, the associated formal parameters may be preserved over calls to the procedures. Thus, interreaching edges are created at the corresponding call sites: interreaching edge (9, 10) at the call to procedure P1 in Main, interreaching edge (7, 8) at the call to procedure P2 in Main, and interreaching edge (3, 4) at the call to procedure P2 in procedure P1. The addition of the binding and interreaching edges completes the construction of the IFG.

4.3 Step 3: Propagate the Local Information to Obtain Interprocedural Information

After the first two steps are completed, the IFG, with local information attached to its nodes, is available. The next step is to propagate the local information throughout the IFG to obtain interprocedural reachable use sets for each node in the graph. Interprocedural reachable use sets represent the upward exposed uses of nonlocal variables located in other procedures that can be reached from the beginning and end of each region of code represented by nodes in the graph. These interprocedural reachable uses are computed by propagating the UPEXP[n] sets backward throughout the graph as far as they can be reached, while taking into account the calling context of the called procedures.

$\mathrm{IN}_{\mathrm{Use}}$ and $\mathrm{OUT}_{\mathrm{Use}}$ for nodes in the IFG are computed in two phases to preserve the calling context of called procedures. Consider the propagation of the UPEXP for node 10 in Figure 5, where UPEXP[10] consists of the use of X in $s3$. If UPEXP[10] is propagated backward in the IFG, it would reach, among others, nodes 6, 4, 2, 1, and 7, meaning that $s3$ is reachable from the call site to P2 in $s4$. However, there is no control path through the program for which statement $s3$ can be reached from this call site. The problem occurs when this use is propagated over the call binding edge (7, 1), since this edge does not match the return context. However, the use must be propagated to nodes 3, 5, and 9, since this path does match the return context. To solve this problem, the propagation is performed in two phases. In the first phase, information propagating to the exit node is not computed, but all other information is allowed to flow across the call binding edges. In the second phase, information reaching the exit node is further propagated using the interreaching edge, but no information is propagated over the call binding edges. This two-phase propagation preserves the calling context of called procedures and ensures that only possible control paths in the program are considered. Therefore, we first process only the entry, call, and return nodes, and propagate the uses that can be reached in called procedures over the call binding edges, the reaching edges, and the interreaching edges. Next, we propagate the uses that can be reached in calling procedures over the return binding edges, the reaching edges, and the interreaching edges. Propagation must be restricted to these edges to prevent traversal of paths through the IFG that do not represent control paths through the program. Since we saw in the above example that propagation over the call binding edges is a problem, the interreaching edges facilitate the propagation of the uses to the call and entry nodes without traversing the call binding edges.

The interprocedural reachable use problem is formulated as a simple distributive data-flow problem [Kildall 1973] whose lattice of solution is "can be reached." A use can be reached from a point in the program if there is a path in the IFG, over return binding, reaching, and interreaching edges, from the node representing the point to the node where UPEXP[n] contains the use. The greatest fixed point of the following data-flow equations captures this fact. In phase one of the propagation, successors of a node **n** consist of the immediate successors of **n** that are entry, call, or return nodes. Thus, only the entry, call, and return nodes are processed in phase one. In phase two, successors of a node **n** consist of the immediate successors of **n** over all edges except the call binding edges. Thus, processing includes all nodes, but no information is propagated over the call binding edges.

The following sets of data-flow equations are used to compute the $\mathrm{IN}_{\mathrm{Use}}[\mathrm{n}]$ and $\mathrm{OUT}_{\mathrm{Use}}[\mathrm{n}]$ sets:

$$\mathrm{OUT}_{\mathrm{Use}}[\mathrm{n}] = \mathrm{OUT}_{\mathrm{Use}}[\mathrm{n}] \cup {}_{\mathrm{s}}\mathrm{IN}_{\mathrm{Use}}[\mathrm{s}], \qquad \text{successors s of n},$$
$$\mathrm{IN}_{\mathrm{Use}}[\mathrm{n}] = \mathrm{OUT}_{\mathrm{Use}}[\mathrm{n}] \cup \mathrm{UPEXP}[\mathrm{n}].$$

To incorporate the results of phase one's computation during phase two, the $\mathrm{OUT}_{\mathrm{Use}}[\mathrm{n}]$ computed in phase one is required in computing the new $\mathrm{OUT}_{\mathrm{Use}}[\mathrm{n}]$. The equations are solved using procedure *Propagate* given in Figure 6. The parameters to *Propagate*, N and E, indicate which nodes and edges, respectively, are considered in the propagation.

Consider again the example in Figure 5. During the third step in the algorithm, the UPEXP sets are propagated throughout the IFG to get the interprocedural reachable uses. We use $\mathrm{IN}_{\mathrm{Use}}$ and $\mathrm{OUT}_{\mathrm{Use}}$ to denote the interprocedural uses that are reachable before and after the control points, respectively, and we attach these sets to the nodes in the IFG. Consider node 3 in procedure P1, which represents the call to procedure P2. The uses of variables that are bound to Y and can be reached from this point in the program consist of the uses of X in $s3$ and $s5$, the use of Y in $s8$, and the uses of Z in $s9$. The use in $s8$ is reached over the interreaching edge since Y is preserved over the call to procedure Z, the uses in $s9$ are reached over the call to procedure P2, and the uses in $s3$ and $s5$ are reached over returns to Main. Likewise, our algorithm propagates the DEF sets throughout the IFG to get the interprocedural reaching definitions that are attached to nodes in the IFG. We use $\mathrm{IN}_{\mathrm{Def}}$ and $\mathrm{OUT}_{\mathrm{Def}}$ to denote these interprocedural reaching definitions. The table included in Figure 5 gives the $\mathrm{IN}_{\mathrm{Use}}$ and $\mathrm{OUT}_{\mathrm{Use}}$ sets, along with the $\mathrm{IN}_{\mathrm{Def}}$ and the $\mathrm{OUT}_{\mathrm{Def}}$ sets, that are computed during the propagation step.

```
procedure Propagate(N, E)
input       N: set of node types to be processed
            E: set of edge types to be processed
begin
    while data flow changes do
        for each node n of type N do
            for each node s that is a successor over E of n do
                $OUT_{Use}[n] = OUT_{Use}[n] \cup IN_{Use}[s]$
                $IN_{Use}[n] = OUT_{Use}[n] \cup UPEXP[n]$
            endfor
        endfor
    endwhile
end Propagate
```

Fig. 6. Procedure for propagating data-flow information.

Procedure *Propagate* iterates over the subset of the nodes and edges in the IFG that are specified in the call until the data-flow sets stabilize. Cycles can occur in an IFG within a subgraph or among subgraphs. In particular, cycles occur (1) within individual subgraphs due to loops in a procedure, (2) in the interconnections of subgraphs because the return from one procedure reaches the call to another, and (3) in programs with recursive procedures. Cycles of types (1) and (3) in an IFG also produce cycles in the subset of the IFG nodes and edges processed by procedure *Propagate* and, thus, require iteration. Cycles of type (2) require no iteration since the subset of IFG nodes and edges processed by procedure *Propagate* contains no cycles.

A cycle of type (1) occurs within a subgraph for a procedure if a call site is contained in a loop in the procedure and if there is also an interreaching edge connecting the call and return nodes. This type of cycle consists of call and return nodes, and reaching and interreaching edges, contained entirely within the subgraph. A subset of nodes for the IFG in either phase one or phase two of *ComputeChains* can contain a cycle. Thus, iteration over the nodes in the IFG is required by procedure *Propagate* to compute the data sets. An example of this type of cycle is given in Figure 7, where a program segment and its associated IFG nodes are shown. In the example, the **for** loop contains calls to both procedures A and B, where variable X is passed as an actual parameter. Nodes 1 and 2 represent the call to and return from procedure A, respectively, while nodes 3 and 4 represent the call to and return from procedure B, respectively. To demonstrate this type of cycle, we assume that the parameters associated with X in procedures A and B may be preserved by calls to the procedures, which accounts for the interreaching edges (1, 2) and (3, 4) connecting the call and return nodes. Reaching edge (2, 3) indicates that any definitions of variable X that reach the return from procedure A also reach the call to procedure B. Likewise, reaching edge (4, 1) indicates that any definition of variable X reaching the return from procedure B also reaches the call to procedure A.

Cycles of type (2) can occur in the interconnections of subgraphs when the return from one procedure reaches the call to another but is not contained in a loop in the procedure. This type of IFG results in a reaching edge from the

```
for i := 1 to n do
    A(X);
    B(X);
endfor
```

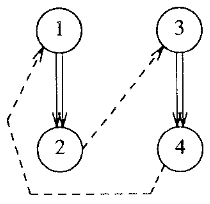


Fig. 7. Example of a cycle of type (1).

return node to the call node. Although there is a cycle in the IFG, the subset of IFG nodes and edges processed by procedure *Propagate* in either phase one or phase two is cycle free. To illustrate this type of cycle, consider the program and its IFG given in Figure 5. The path in the IFG through nodes 7, 1, 2, 4, 6, 10, 7 is an example of a cycle of this type. However, phase one does not include nodes 2 or 6 or edges (2, 4), (4, 6) or (6, 10), and consequently, no cycle is processed by procedure *Propagate*. Since edge (7, 1) is not included in the processing in phase two, no cycle exists among the subset of nodes and edges, and thus, no iteration is required for *Propagate*.

The presence of recursive procedures may cause cycles of type (3). To illustrate, consider the program and its IFG given in Figure 8, which differs from the program in Figure 1 in that procedure P2 calls procedure P1, causing procedures P1 and P2 to be mutually recursive. Note the additional nodes representing the call to and return from procedure P1 (i.e., 11 and 12), the binding edges (i.e., (11, 5) and (6, 12)), the reaching edges (i.e., (1, 11) and (12, 2)), and the interreaching edge (i.e., (11, 12)). The path through nodes 5, 3, 1, 11, 5 is a cycle in the IFG that is processed by procedure *Propagate* during phase one of *ComputeChains*. Thus, iteration is required in the procedure *Propagate* to compute the data sets.

4.4 Step 4: Compute the Definition-Use Chains

The reachable use information that is computed for the IFG is used with local DEF sets to compute the interprocedural definition-use chains. Interprocedural definition-use chains are computed by considering DEF sets for call and exit nodes associated with each procedure. If **d** is a definition in DEF[n], where n is either a call or return node, then the interprocedural definition-use chain of **d** consists of the elements in $\mathrm{OUT}_{\mathrm{Use}}[n]$. If **d** is in DEF[n] for more than one n associated with the procedure, then the interprocedural definition-use chain is the union of the $\mathrm{OUT}_{\mathrm{Use}}$ sets for all n where DEF[n] contains **d**.

Again, refer to our running example whose completed IFG is shown in Figure 5. We compute the definition-use chains of each interprocedural definition in the program. For example, the interprocedural definition-use chain of the definition of Y in $s6$ is the set of uses that can be reached from node 3 or $\{s3, s5, s8, s9\}$.

4.5 Computation of Use-Definition Chains

An analogous technique for propagation allows our algorithm to be modified and then used to compute the interprocedural use-definition chains. The

```
program Main
  s1 read(X);
  s2 P1(X); {nodes 9, 10}
  s3 if X>20 then X:=X+6,
  s4 P2(X); {nodes 7, 8}
  s5 write(X);
end Main;
```

```
procedure P1(Y) {node 5}
  s6 if Y<10 then Y := Y + 4;
  s7 P2(Y); {nodes 3, 4}
  s8 if Y<25 then Y := Y + 5;
end P1; {node 6}
```

```
procedure P2(Z) {node 1}
  s9 if Z < 10 then P1(Z);
        {nodes 11, 12}
  s10 if Z > 10 then Z := Z + 2;
end P2; {node 2}
```

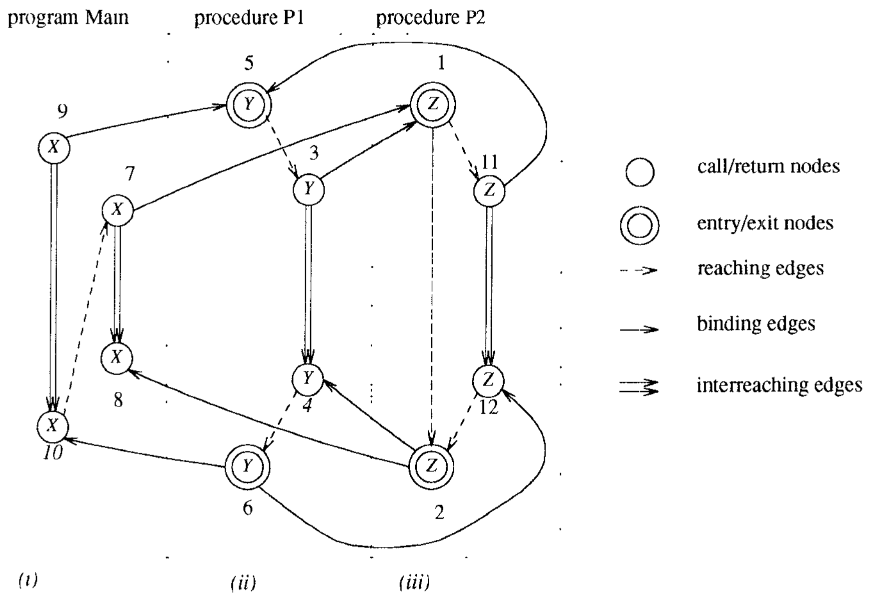


Fig. 8. Recursive program and its interprocedural flow graph.

computation of interprocedural use-definition chains requires changes to Steps 3 and 4 of the *ComputeChains* algorithm. Additionally, procedure *Propagate* is also changed to reflect the different data-flow information being computed. Steps 3 and 4 for computing the use-definition chains are shown in Figure 9. Here, interprocedural reaching definitions are obtained by propagating the local definition information (i.e., the DEF sets) throughout the program using the IFG. The propagation of the definitions is in the forward direction throughout the IFG, taking into account the calling context of called procedures. Thus, propagation Step 3 in algorithm *ComputeChains* is altered to reflect the node sets required to preserve the calling context of called procedures, while procedure *Propagate* is altered to reflect the forward direction of the data flow. Like the IN_{Use} and OUT_{Use} sets used to compute the reachable use information, $IN_{Def}[n]$ and $OUT_{Def}[n]$ refer to the reaching definition sets associated with node n. In order to preserve the calling context when the data-flow equations represent a backward problem, the exit node in the first phase and the call binding edge in the second phase are not considered. Similarly, to preserve the calling context, when the data-flow

```
declare UDC: array of use-definition chains

/* Step 3:  IFG propagation to obtain global information */
    for each node n in G do                          /* initialization */
        $IN_{Def}[n] = \phi$
        $OUT_{Def}[n] = DEF[n]$
    endfor
    NODESET := {call, return, exit nodes in G}       /* phase 1 */
    EDGESET := {all edges in G}
    Propagate(NODESET, EDGESET)
    NODESET := {all nodes in G}                      /* phase 2 */
    EDGESET := {call binding, reaching, interreaching edges in G}
    Propagate(NODESET, EDGESET)
/* Step 4:  interprocedural definition-use chains computation */
    for each $P_i \in P$ do
        for each interprocedural use u in $P_i$ do
            $UDC[u] := \phi$
            if $u \in UPEXP[return_x^{P_i->Q}]$ then $UDC[u]:=UDC[u] \cup IN_{Def}[return_x^{P_i->Q}]$
            if $u \in UPEXP[entry_f^P]$ then $UDC[d] := UDC[d] \cup IN_{Def}[entry_f^P]$
        endfor
    endfor
```

Fig. 9. Partial interprocedural use-definition chains algorithm.

equations represent a forward problem, the entry node in the first phase and the return binding edge in the second phase are not considered. Thus, in phase one of the propagation, call, return, and exit nodes and all edges are considered, while in phase two, all nodes are considered, but edges are restricted to call binding, reaching, and interreaching. The data-flow equations for reaching definitions are

$$IN_{Def}[n] = IN_{Def}[n] \cup {}_{p}OUT_{Def}[p], \qquad \text{where p is a predecessor of n,}$$
$$OUT_{Def}[n] = IN_{Def}[n] \cup DEF[n].$$

Procedure *Propagate* is changed to reflect the change in data-flow equations. After the reaching definitions are computed, analogous to Step 4 of the algorithm, the use-definition chains are found by considering the UPEXP at each node in the IFG and the definitions that reach that node.

Consider the effects of Step 4 on the example in Figure 4 where we compute interprocedural definition-use chains. For example, the interprocedural use-definition chain of the use of Y in *s*6 is the set of definitions that reach node 5 or {*s*1}.

4.6 Handling Global Variables

Since the names of global variables do not change within a program, only a single node is required at each entry, exit, call, and return point of a procedure to represent information about the global variables in that proce-

```
procedure Propagate'(N, E)
input      N: set of node types to be processed
           E: set of edge types to be processed
begin
     while data flow changes do
          for each node n of type N do
               for each node p that is a predecessor over E of n do
                    $IN_{Def}[n] = IN_{Def}[n] \cup_p OUT_{Def}[p]$
                    $OUT_{Def}[n] = IN_{Def}[n] \cup DEF[n] - KILL[n]$.
               endfor
          endfor
     endwhile
end Propagate'
```

Fig. 10. Procedure for propagating global reaching definition information.

dure [Callahan 1988]. DEF and UPEXP sets are computed for global variables and attached to the appropriate global nodes. We create reaching and interreaching edges in the usual way to abstract possible control information in the program that is related to the global variables. A bit vector is attached to each of these edges indicating which global variables may be preserved by execution through the associated regions of code in the program. Unlike the nodes representing reference parameters in the IFG, global nodes represent more than one variable. Thus, a KILL set, indicating which global variables are not preserved on execution through the region represented by the node, is required for the propagation phase in much the same way that a KILL set is attached to each basic block in the intraprocedural flow graph. For reaching definitions, IN_{Def} and OUT_{Def} sets are computed at each node in the IFG. These sets are also computed in two phases to preserve the calling context of called procedures, as is done for reaching definitions of parameters. In phase one of the propagation, predecessors must be call, return, or exit nodes, and all edges are considered. In phase two of the propagation, all nodes are considered, but only call binding, reaching, and interreaching edges are considered. The data-flow equations for reaching definitions of global variables are

$$IN_{Def}[n] = IN_{Def}[n] \cup {}_pOUT_{Def}[p], \qquad \text{where p is a predecessor of n},$$
$$OUT_{Def}[n] = IN_{Def}[n] \cup DEF[n] - KILL[n].$$

Procedure *Propagate'*, given in Figure 10, reflects the required changes in the data-flow equations.

4.7 Complexity Analysis

In this section we analyze both the space complexity of the IFG and the time complexity of the *ComputeChains* algorithm. The size of the IFG can be expressed in terms of the parameters given in the Table I.

Table I Parameters Expressing the Size of the Interprocedural Flow Graph

C	Number of call sites per procedure
G	Number of global variables in the program
P	Number of procedures in the program
F	Number of formal parameters per procedure
A	Number of actual parameters per call site
Ac	Total number of actual parameters per procedure

Our analysis shows that the size of an IFG is polynomial in the size of the program when we consider both formal parameters and global variables. We discuss our analysis in two parts. First we describe our analysis per procedure for an IFG subgraph containing only formal parameters. Then, we present a similar analysis for global variables.

For each procedure, the IFG subgraph restricted to formal parameters contains 2F + 2AC nodes, an entry/exit pair for each formal parameter, and a call/return pair for each actual parameter at each call site. Since there are two binding edges for each actual parameter at each call site, and a maximum of one interreaching edge for each actual parameter at each call site, the number of binding edges is 2AC + AC. At worst, there are reaching edges from entry nodes to exit nodes for each formal parameter, edges from entry node to call node for each actual parameter at the call site, an edge from return to exit for each formal parameter and edges from each call site to each other call site for each actual parameter at the destination call sites. Thus, the number of reaching edges is given by F + AC + FC + AC(C − 1). The total number of edges is the sum of the binding edges and the reaching edges, which simplifies to AC(C + 3) + F(C + 1). An IFG subgraph is the sum of its nodes and edges, or AC(C) + C(5A + F) + 3F, which is O(Ac* C), since AC is Ac, the total number of actual parameters per procedure, and F is a small constant per procedure [Cooper and Kennedy 1988].

Although the size of an IFG subgraph with respect to formal parameters is quadratic in the size of the procedure, we do not expect this to be the case in practice. Our experimental programs average 6.7 formal parameters per procedure and 4.1 actual parameters per call site. In the worst case, where each statement is a call site, an IFG subgraph would have 602 nodes and 44,527 edges. However, in practice, call sites are much less frequent. In our experimental sample, call sites constitute only 3.9 percent of the statements. Thus, we expect only 36 nodes and, at worst, 108 edges. Moreover, the worst-case edge analysis assumes that edges reach from site to site in all cases. In our experimental sample, each procedure had on average only 28.8 edges.

Next, consider the nodes and edges required in an IFG subgraph to represent information on global variables for a procedure. Each procedure has one entry/exit pair, and each call site has one call/return pair. Thus, the number of nodes for globals in an IFG subgraph is 2 + 2C. At each call site, there is a call binding and a return binding edge, and there is a single interreaching edge. There are also reaching edges to each call site from entry and to exit nodes, from entry to exit, and at most one edge from every call site to every other call site. Adding all edges and simplifying give $C^2 + 4C + 1$ or $O(C^2)$. Thus, the size of the IFG for globals is also quadratic in the size of the procedure.

The time complexity of our *ComputeChains* algorithm is determined by considering each of the four steps. In the first step, the creation of the graph requires one visit to each of **n** nodes in the IFG. The last step in the algorithm is performed by considering the definitions in each DEF set and combining the appropriate OUT_{Use} sets to get the interprocedural definition-use chains. This last step also requires one visit to each node during the computation. In Step 2, the preserved information that is required for the interreaching edges is computed. For programs with no recursion, this computation is linear in the number of nodes in the IFG [Callahan 1988]. The propagation of the local information throughout the graph is accomplished in Step 3. As with many data-flow analysis problems, the propagation in Step 3 is $O(n^2)$ in the worst case. Although the *ComputeChains* algorithm is $O(n^2)$, it is a monotone, distributive and rapid data-flow analysis framework [Kam and Ullman 1976]. Thus, the algorithm requires $\mathbf{d} + 3$ visits to each of the **n** nodes, where **d** indicates the loop connectedness for a depth-first ordering of the nodes in the graph. As each node is visited, set operations are performed on sets representing information about the interprocedural definitions in the program. Although the size of these sets is potentially as large as the number of definitions in the entire program, we restrict members of the set to those statements that actually define a value that is used in another procedure in the program. In our experimental sample, we found that the maximum size of the sets required was 149, while the average was much lower.

5. DEFINITION-USE CHAINS IN THE PRESENCE OF ALIASING

In Section 4 we have described our algorithms for finding interprocedural data-flow chains for programs that are alias-free. In this section we present two approaches for handling programs with aliasing due to reference parameters and global variables. Our first approach uses a technique by Horwitz, Reps, and Binkley [1990] to unalias a program, and then applies the *ComputeChains* algorithm to the unaliased version of the program. Their technique to unalias a program uses a program's activation tree to create a new copy of the procedure for each different alias configuration. Unaliasing the program in this way is potentially exponential in the number of parameters passed to a procedure. However, data-flow chains computed with this unaliased version of the program are precise since the control flow of the procedure is considered with each different alias configuration.

Program Main in Figure 11 illustrates this technique. Assume that X1 and X2 are local to Main. At the first call to procedure P in *s*2, both X1 and X2 are passed as actual parameters, and no alias is introduced into P. However, in the second call to procedure P in *s*4, X1 is passed in both actual parameter positions, and thus, an alias is introduced at this call site. To unalias the program, another version of procedure P (procedure P′) is created that

```
program Main
  s1  read(X1, X2),
  s2  P(X1, X2), {nodes 1,2,3,4}
  s3  write(X1, X2);
  s4  P'(X1), (*P(X1, X1),*)
  s5. write(X1, X2),
  end Main,
```

```
procedure P(Y, Z)  {nodes 7, 9}
  s6  if Y<10 then Y = Y + 4,
  s7  else Z = Z + 6,
end P,  {nodes 8, 10}
```

```
procedure P'(YZ)  {node 11}
  s8  if YZ<10 then YZ = YZ + 4,
  s9  else YZ = YZ + 6,
end P',  {node 12}
```

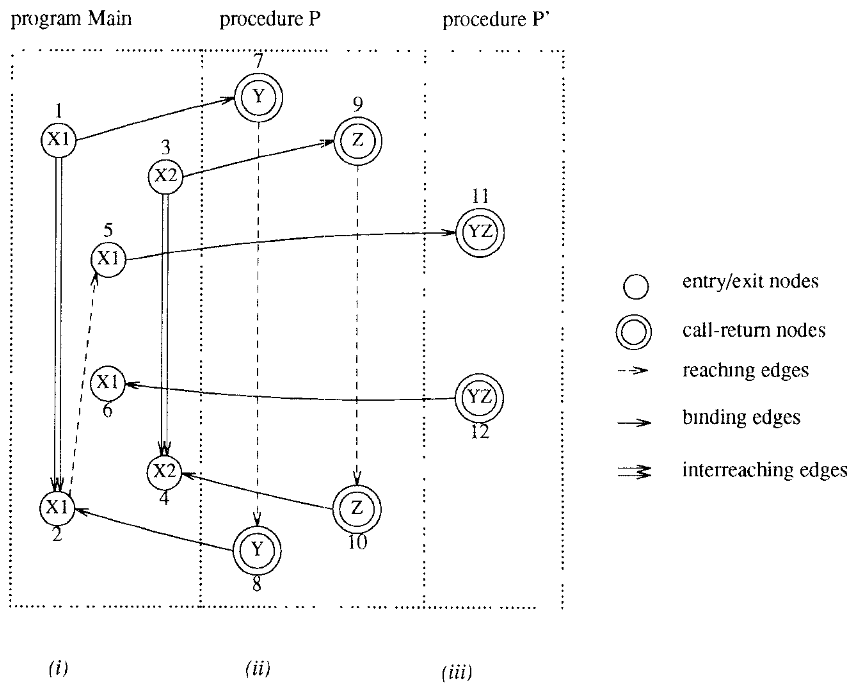


Fig. 11. Example program and its interprocedural flow graph's subgraphs illustrating computation of data-flow chains in an alias environment. Since the program is unaliased with this technique, the data-flow chains are precise.

contains only one formal parameter, which we call XY. Then, all occurrences of formal parameters X and Y are replaced with XY. In the original version of procedure P, both formal parameters Y and Z are preserved on a path from the entry of procedure P to its exit. Thus, definitions of the corresponding actual parameters X1 and X2 can reach over the call to P. However, on the call to procedure P′, there is no path on which formal parameter YZ is preserved. With this additional copy of procedure P, we get precise definition and use information at the expense of the additional cost of the unaliased version of the program.

Since unaliasing a program may be prohibitive due to the increase in code size, we provide an alternative solution that provides safe but somewhat imprecise data-flow chains in the presence of aliasing. We incorporate alias information into the construction of the IFG by altering Steps 1 and 2 of

ComputeChains. The last two phases of *ComputeChains* remain unchanged since they are applied to the alias version of the IFG. We assume that alias information has been computed for the program using an algorithm such as the one by Cooper and Kermedy [1988] to compute alias pairs for global variables and reference parameters. Then, when we construct the IFG subgraph for a procedure, we combine corresponding call nodes for actual parameters that are aliases of each other at that call site. We also combine the corresponding return nodes for the aliased variables. By combining call and return nodes where aliasing occurs, we introduce some imprecision into our analysis and may get some spurious data-flow pairs. However, our technique results in safe information since no valid data-flow pairs are missed.

To illustrate, consider the program in Figure 12, which is the same as the program in Figure 11 except that, instead of unaliasing the program, we have altered the IFG to account for the known alias pairs. At call site P(X1, X1) in $s4$, there is a call and return node for each actual parameter, and these parameters are bound to formal parameters Y and Z in procedure P. However, since the call nodes represent a call site where an alias is introduced, the nodes are combined to get a single node with two edges: one edge from the combined node to each of the corresponding formal parameter nodes. Since we already have the reaching edges computed in the alias-free environment, there will be spurious definitions that reach the return from procedure P represented by return node. In cases where the reaching information changes as a result of aliased variables, imprecision in the form of extra data-flow information results. Consider the definition of X1 in $s1$. During propagation, this definition reaches the combined return node 6–8, which indicates that there is a definition-use pair from X1 in $s1$ to the use of X1 in $s5$. However, we can see by inspecting the program that the value of X1 is killed on either path through procedure P when Y and Z are aliased and, thus, can never reach the use in $s5$.

In an alias-free environment, global variables and formal parameters are handled separately. However, if global variables are aliased to formal parameters, the information must be combined. Any global variable that is passed as a parameter is treated as a parameter rather than a global. Thus, we create entry and exit nodes for that global for each procedure entry, and we create call and return nodes for that global at each call site. We also remove information about this global from the global node. Unlike other techniques that create nodes at each entry, exit, call, and return for each global variable [Horwitz et al. 1990], we create these additional nodes only when the global is aliased to a formal parameter.

6. EXPERIMENTAL RESULTS

We have implemented a prototype that incorporates our techniques for computing interprocedural definition-use chains. Our prototype is written in C and was developed using a Sun-4 Workstation. We augmented the Free Software Foundation,© Inc., GNU C compiler to gather intraprocedural data-flow information to enable both construction of the IFG and interprocedural

```
program Main
  s1: read(X1, X2),
  s2: P(X1, X2); {nodes 1, 2, 3, 4}
  s3: write(X1, X2),
  s4: P(X1, X1); {nodes 5, 6, 7, 8}
  s5: write(X1, X2);
end Main,
```

```
procedure P(Y, Z) {nodes 9, 11}
  s6: if Y<10 then Y := Y + 4;
  s7: else Z := Z + 6;
end P; {nodes 10, 12}
```

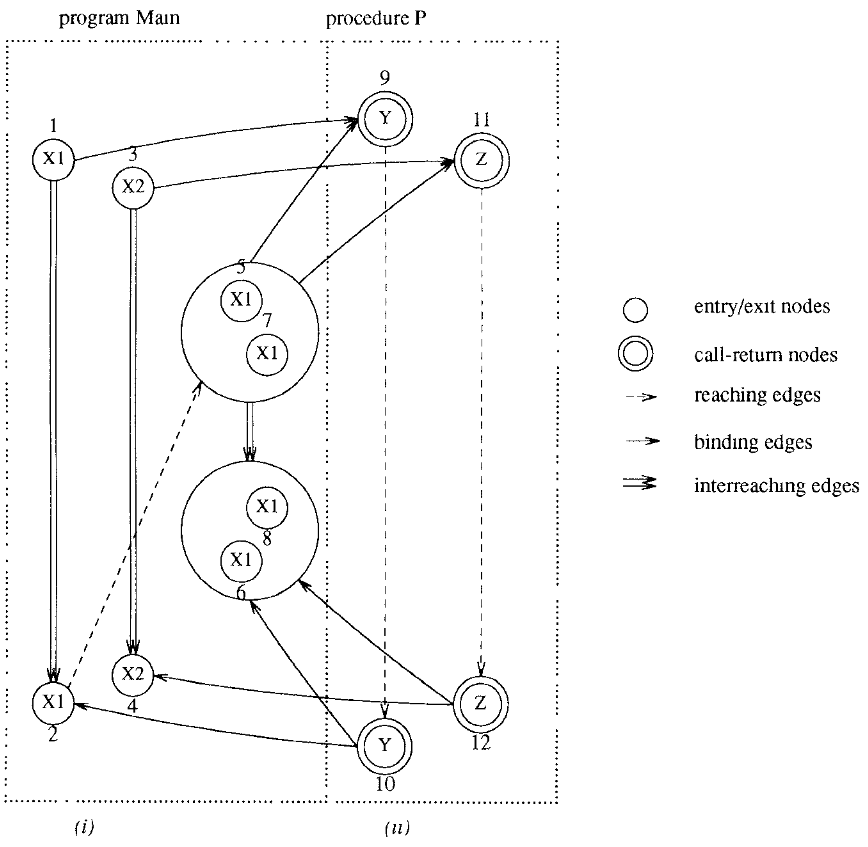


Fig. 12. Example program and its interprocedural flow graph's subgraph, illustrating handling of an environment with aliasing. This technique produces safe data-flow chains.

analysis. Our prototype inputs the intraprocedural information for a group of procedures, constructs an IFG, and performs interprocedural analysis. The input to the prototype is a group of procedures that may not represent an entire program. Thus, we handle incomplete programs and show only the relationship among interacting procedures. Our ability to handle incomplete programs facilitates interprocedural data-flow testing during the integration phase of software development. We consider the effects of formal reference parameters and global variables on interprocedural information.

Table II. Experimental Sample Set of FORTRAN Programs

Program	Number of statements	Number of procedures	Description
Mach_params	88	6	Obtains machine-independent parameters
Basic	134	7	Interpreter for Basic-style math expressions
561	197	16	Heap procedures for use in table maintenance
Whetstone	201	7	Whetstone benchmark program
Mem	226	6	Simulates a variable partitioned memory-management scheme
Dlock	366	8	Implements Dijkstra's Banker's algorithm for deadlock detection
500	403	3	Computes the minimization of unconstrained multivariate functions
547	503	4	Routines for discrete cubic spline interpolation/smoothing
Slice	561	7	Computes the eigenvalues on an interval in a tridiagonal matrix
604	593	11	Routines for calcultion of an extremal polynomial
Inverse2	603	14	Solves a general $N \times N$ system of linear equations
618	607	12	Routines for estimating sparse Jacobian matrices
576	696	8	Computes the solution for ax = b
Dag	753	4	Computes the diagonalization of a complex Hessenberg matrix
Mpower	754	7	Computes several eigenpairs using the power method
534	853	7	Stiff differential equations integrator
522	923	6	Congruence techniques to solve integer systems of linear equations
525	1,401	18	Adaptive smooth curve fitting
684	1,672	24	C1, C2 interpolation on triangles with quintics and nonic
686	1,697	25	Updating QR
606	1,781	25	Interactive tree package
683	2,115	28	Exponential integral of complex argument
527	2,524	33	Implementation of the generalized marching algorithm
624	2,526	37	Triangulation/interpretation of arbitrarily distributed points in the plane
623	2,580	41	Interpolation on the face of a sphere
541	3,922	29	Computes the solution of separable elliptic partial differential equations
524	4,431	129	Multiprecision arithmetic package
586	6,876	99	Package to solve large sparse linear systems/adaptive accelerated interactive methods

We experimented with a set of FORTRAN programs of varying sizes to determine the sizes of both the IFGs and the propagated sets, relative to the worst-case size complexity of the algorithms. Table II lists the FORTRAN programs, the number of statements and procedures in each program, and a brief description of each program; numbered programs were taken from the Collected Algorithms of *ACM Transactions on Mathematical Software*.

Table III. Sizes of the Interprocedural Flow Graphs (IFGs) for the Experimental Sample

Program	Number of statements	Number of procedures	Number of global variables	Number of formal parameters	Number of actual parameters	Size of IFG	IFG/ statements
Mach_params	88	6	14	18	20	237	2.7
Basic	134	7	0	15	54	228	1.7
561	197	16	9	67	100	954	4.8
Whetstone	201	7	4	11	64	686	3.4
Mem	226	6	4	2	10	153	0.7
Dlock	366	8	5	20	42	428	1.2
500	403	3	4	19	14	132	0.3
547	503	4	4	20	12	138	0.3
Slice	561	7	7	55	91	682	1.2
604	593	11	11	84	112	1,179	2 0
Inverse2	603	14	15	64	178	1,586	2.6
618	607	12	1	89	92	785	1.3
576	696	8	4	54	217	1,463	2.1
Dag	753	4	11	17	25	279	0.4
Mpower	764	7	14	46	148	1,021	1.3
534	853	7	3	42	47	412	0.5
522	923	6	2	51	98	782	0.8
525	1,401	18	7	872	84	808	0.6
684	1,672	24	5	147	329	2,269	1.4
686	1,697	40	6	231	737	5,347	3.1
606	1,781	25	0	200	1,132	5,802	3.3
683	2,115	28	14	152	380	3,076	1.5
527	2,524	33	9	305	503	3,908	1.5
624	2,526	37	0	272	574	3,297	1.3
623	2,580	41	1	306	590	4,417	1.7
541	3,922	29	13	362	393	3,268	0.8
524	4,431	129	12	314	2,341	17,587	3 9
586	6,876	99	13	632	2,285	16,292	2 4

We instrumented each program to ascertain the actual number of nodes and edges in the IFG for that procedure for both formal parameters and global variables. This total gives the actual size of the IFG and these results are given in Table III. For each program we list the number of statements, procedures, global variables, formal and actual parameters, size of the IFG, and ratio of the IFG to the number of statements. The size of the IFG shown here includes nodes and edges for both global variables and formal parameters. The ratios of the IFG to the number of statements range from less than 1 to almost 5 and are independent of the program size. For example, the biggest ratio is for program 561, having 197 statements; whereas one of the smallest ratios is for program 541, having 3922 statements. Another important implication of our experimentation is that the relative sizes of the IFG did not increase with program size. Thus, in practice, we expect the size of the IFG to be proportional to the size of the program for both global variables and formal parameters.

In addition to experimentation to find the expected size of an IFG, we also wanted to determine the space requirements for the data-flow sets. We instrumented the program to count interprocedural reaching definitions at call sites and procedure exits. In the worst case, these sets can be as large as the number of statements in the program, which may be prohibitive. However, for our experimental sample, we found that the maximum number of interprocedural definitions that reached a call or exit site was 20 for program 604. This maximum is relatively small compared to the number of statements in the program. We also found that the average number of interprocedural definitions that reach a site was 1.95. Thus, we expect these sets to be small in practice.

7. RELATED WORK

A number of data-flow analysis techniques have been developed that compute interprocedural dependency information. None of these techniques efficiently computes adequate information to solve the reaching definitions and reachable uses problems that we address. Some existing flow-insensitive[1] data-flow analysis techniques [Banning 1979; Barth 1978; Cooper and Kennedy 1988; Lomet 1977] provide summary data-flow information for determining the local effects of called procedures at call sites. These techniques do not provide information about the locations of interprocedural definitions and uses in other procedures in the program. A flow-sensitive technique that processes nonrecursive procedures in reverse invocation order [Allen et al. 1987] incorporates the abstracted information about called procedures at call sites to obtain the local reaching information. The technique requires that a procedure be processed only after those that it calls have been processed, which imposes an ordering on the procedure processing. This order restriction results in a penalty when changes are made in a procedure, for it causes the reanalysis of those procedures directly or indirectly dependent on the changed procedure. Also, the technique does not compute the locations of the definition-use chains across procedure boundaries and cannot handle recursive procedures with the ordering restriction.

The *program summary graph* developed by Callahan [1988] provides flow-sensitive interprocedural data-flow information that solves the interprocedural KILL, MOD, and USE problems. For example, the KILL of each formal parameter in a procedure is a Boolean that indicates whether the variable is redefined along all paths by a call to the procedure. An iterative technique uses the paths through the program summary graph to compute the KILL for the formal parameters in each procedure. Thus, for the program in Example 1, this technique will determine that neither variable Y in P1 nor variable Z in P2 is KILLed by calls to those procedures. However, this technique cannot be used to compute the interprocedural definition-use pairs, since the program summary graph does not contain information about the locations of the

[1]Interprocedural data-flow information is *flow-sensitive* if the control flow of called procedures is used in the computation.

definitions and uses in the program. Furthermore, the structure of the graph does not allow for the preservation of the calling context of called procedures.

A third related technique [Horwitz et al. 1990] provides interprocedural slicing of nonrecursive programs using the *system dependence graph*, which combines the dependence graphs for each procedure to provide a representation of the program. It also handles the problem of preserving the calling context of called procedures in order to provide more precise slices. The algorithm consists of two phases, each of which visits a subset of the nodes in the entire graph. It is possible to find the required definition-use chains by computing the *forward* slice for each interprocedural definition in the program and then extracting the reachable uses from the information in the slice. Clearly, this method requires that the slicing algorithm be run for each interprocedural definition in the program. Our algorithm also consists of two phases, each of which visits a subset of the nodes in the IFG. However, all of the required data dependency information for procedures is computed by one application of our algorithm, in contrast to the slicing algorithm, which requires one repetition of the algorithm for each interprocedural definition in the program.

Finally, use of the *super graph* [Myers 1981] or in-line substitution to compute the interprocedural definition-use and use-definition chains is prohibitive for large programs. In addition, separate compilation cannot be supported since either the code or the flow graph of each procedure must be available during the analysis. A further restriction of in-line substitution is that recursive procedures cannot be analyzed.

8. CONCLUSIONS

In this paper we have presented techniques for computing interprocedural definition-use chains. A graph structure that abstracts data-flow information for each procedure has been utilized for the efficient propagation of the definition-use information. A data-flow analysis algorithm has been developed to compute reaching definitions and reachable uses, and constructed in two phases to preserve the calling context of called procedures. The data-flow analysis is performed on a procedure's code without requiring information from other procedures, thus supporting separate compilation. The algorithm also handles recursive procedures and alias pairs.

The technique was implemented in C on a Sun-4 workstation, and experiments were conducted to determine the practicality of the technique. Results from these experiments indicate that the ratio of the size of the IFG, measured by the total number of nodes and edges, to the number of statements in a program ranged from less than 1 to 5. The relative size of the IFG did not increase with program size. Another important result indicated that the sizes of the data-flow sets, which could be as large as the number of statements in a program, had 20 elements as the maximum. Experiments also indicate that the average number of interprocedural definitions that reach a site was approximately 2. From these results, the interprocedural technique developed in this paper is a viable technique for integration into software tools. We are currently incorporating this facility in a testing tool that will perform data-flow testing at both the intraprocedural and interprocedural levels.

ACKNOWLEDGMENTS

We wish to thank Priyadarshan Kolte, who augmented the gcc compiler to gather intraprocedural information; G. Regan Varenhorst, who implemented both the IFG and related algorithms; and Gregg Rothermel, who performed the experiments.

REFERENCES

AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1986. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, Reading, Mass.

ALLEN, F. E., BURKE, M., CHARLES, P., CYTRON, R., AND FERRANTE, J. 1987. An overview of the PTRAN analysis system for multiprocessing. In *Proceedings of 1st International Conference on Supercomputing* (June). Springer-Verlag, New York, 194–211.

BANNING, J. P. 1979. An efficient way to find the side effects of procedure calls and aliases of variables. In *Sixth Annual ACM Symposium on Principles of Programming Languages* (Jan). ACM, New York, 29–41.

BARTH, J. M. 1978. A practical interprocedural data flow analysis algorithm. *Commun. ACM 21,* 9 (Sept.), 724–736.

BURKE, M. 1990. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *ACM Trans. Program. Lang. Syst. 12,* 3 (July), 341–395.

CALLAHAN, D. 1988. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Proceeding of SIGPLAN '88 Conference on Programming Language Design and Implementation* (June). *ACM SIGPLAN Not. 23,* 7 (July).

COOPER, K., AND KENNEDY, K. 1988. Interprocedural side-effect analysis in linear time. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation. ACM SIGPLAN Not. 23,* 7 (July).

COOPER, K., KENNEDY, K., AND TORCZAN, L. 1986a. The impact of interprocedural analysis and optimization in the Rn programming environment. *ACM Trans. Program. Lang. Syst. 8,* 4, 491–523.

COOPER, K. D., KENNEDY, K., AND TORCZON, L. 1986b. Interprocedural optimization: Eliminating unnecessary recompilation. In *Proceedings of SIGPLAN '86 Symposium on Compiler Construction. ACM SIGPLAN Not. 21,* 7 (July).

FRANKL, P. G., AND WEYUKER, E. J. 1985. A data flow testing tool. In *ACM Softfair Proceedings* (Dec.). ACM, New York, 46–53.

FRANKL, P. G., AND WEYUKER, E. J. 1988. An applicable family of date flow testing criteria. *IEEE Trans. Softw. Eng. SE-14,* 10 (Oct.), 1483–1498.

HARROLD, M. J., AND SOFFA, M. L. 1989. An incremental data flow testing tool. In *Proceedings of the 6th International Conference on Testing Computer Software* (Washington D.C., May).

HARROLD, M. J., AND SOFFA, M. L. 1991. Selecting data for integration testing. *IEEE Softw. 8,* 2 (Mar.), 58–65.

HORWITZ, S., REPS, T., AND BINKLEY, D. 1990. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst. 12,* 1 (Jan.), 26–60.

KAM, J., AND ULLMAN, J. 1976. Global data flow analysis and iterative algorithms. *J. ACM 23,* 1 (Jan.), 158–171.

KILDALL, G. 1973. A unified approach to global program optimization. In *ACM Symposium on Principles of Programming Languages.* ACM, New York, 194–206.

KOREL, B., AND LASKI, J. 1985. A tool for data flow oriented program testing. In *ACM Softfair Proceedings.* (Dec.). ACM, New York, 35–37.

LASKI, J. W., AND KOREL, B. 1983. A data flow oriented program testing strategy. *IEEE Trans. Softw. Eng. SE-9*, 3 (May), 347–354.

LOMET, D. B. 1977. Data flow analysis in the presence of procedure calls. *IBM J. Res. Dev. 21*, 6 (Nov.), 559–571.

MYERS, E. W. 1981. A precise inter-procedural data flow algorithm. In *Conference Record of the 8th Annual ACM Symposium on Principles of Programming Languages* (Williamsburg, Va. Jan.). ACM, New York, 219–230.

NTAFOS, S. C. 1984. An evaluation of required element testing strategies. In *Proceedings of 7th International Conference on Software Engineering* (Mar.). IEEE, New York, 250–256.

TAHA, A. M., THEBUT, S. M., AND LIU, S. S. 1989. An approach to software fault localization and revalidation based on incremental data flow analysis. In *Proceedings of COMPSAC 89* (Sept.). IEEE, New York, 527–534.

Received July 1990; revised March 1993; accepted April 1993