

Representation and Analysis of Software

Mary Jean Harrold
Georgia Tech

Gregg Rothermel
Oregon State University

Alex Orso
Georgia Tech

1 Introduction

This paper presents some basic techniques for representation and analysis of software. We use the term *program* to refer to both procedures and monolithic programs.

2 Control Flow Graphs

A *control flow graph*¹ (CFG) is a directed graph in which each node represents a basic block and each edge represents the flow of control between basic blocks. To build a CFG we first build basic blocks, and then we add edges that represent control flow between these basic blocks.

A *basic block* is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end. We can construct the basic blocks for a program using algorithm `GetBasicBlocks`, shown in Figure 1.

algorithm `GetBasicBlocks`

Input. A sequence of program statements.

Output. A list of basic blocks with each statement in exactly one basic block.

Method.

1. Determine the set of *leaders*: the first statements of basic blocks. We use the following rules.
 - (a) The first statement in the program is a leader.
 - (b) Any statement that is the target of an conditional or unconditional goto statement is a leader.
 - (c) Any statement that immediately follows a conditional or unconditional goto statement is a leader.

Note: control transfer statements such as *while*, *if-else*, *repeat-until*, and *switch* statements are all “conditional goto statements”.
2. Construct the basic blocks using the leaders. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

Figure 1: Algorithm to construct basic blocks for a program.

When we analyze a program’s intermediate code for the purpose of performing compiler optimizations, a basic block usually consists of a maximal sequence of intermediate code statements. When we analyze source code, a basic block consists of a maximal sequence of source code statements. We often find it more convenient in the latter case, however, to just treat each source code statement as a basic block.

¹See Reference [2] for additional discussion of control flow graphs.

algorithm GetCFG

Input. A list of basic blocks for a program where the first block (B_1) contains the first program statement.

Output. A list of CFG nodes and edges.

Method.

1. Create *entry* and *exit* nodes; create edge (*entry*, B_1); create edges (B_k , *exit*) for each basic block B_k that contains an exit from the program.
2. Traverse the list of basic blocks and add a CFG edge from each node B_i to each node B_j if and only if B_j can immediately follow B_i in some execution sequence, that is, if:
 - (a) there is a conditional or unconditional goto statement from the last statement of B_i to the first statement of B_j , or
 - (b) B_j immediately follows B_i in the order of the program, and B_i does not end in an unconditional goto statement.
3. Label edges that represent conditional transfers of control as “T” (true) or “F” (false); other edges are unlabeled.

Figure 2: Algorithm to construct the CFG for a program.

After we have constructed basic blocks, we can construct the CFG for a program using algorithm **GetCFG**, shown in Figure 2. The algorithm also works for the case where each source statement is treated as a basic block. To illustrate, consider Figure 3, which gives the code for program **Sums** on the left and the CFG for **Sums** on the right. Node numbers in the CFG correspond to statement numbers in **Sums**: in the graph, we treat each statement as a basic block. Each node that represents a transfer of control (i.e., 4 and 7) has two labeled edges emanating from it; all other edges are unlabeled.

In a CFG, if there is an edge from node B_i to node B_j , we say that B_j is a *successor* of B_i and that B_i is a *predecessor* of B_j . In the example, node 4 has successor nodes 5 and 12, and node 4 has predecessor nodes 3 and 11.

Figure 4 gives another example program **Trivial** and its CFG. Notice that **Trivial** contains an **if-then-else** statement and a **switch** statement. In the CFG, we have inserted nodes J1 and J2. These nodes, which we call “join” nodes, would not be created by algorithm **GetCFG**; however, we often find it convenient to insert them, to represent places where flow of control merges following a conditional statement. Notice that we represent the **switch** predicate as a node with multiple out-edges – one for each case value. Each edge is labeled with the value to which the switch predicate must evaluate in order for that edge to be taken.

In CFGs built from source code, where we use statements as basic blocks, we do not create blocks for non-executable structure-delimiting statements such as “begin”, “end”, “else”, “endif”, “endwhile”, “case”, “}”, or “{”.

Program Sums

```

1.  read(n);
2.  i = 1;
3.  sum = 0;
4.  while (i <= n) do
5.      sum = 0;
6.      j = 1;
7.      while (j <= i) do
8.          sum = sum + j;
9.          j = j + 1;
10.         endwhile;
11.     write(sum, i);
12.     i = i + 1;
13. endwhile;
14. write(sum, i);
15. end Sums

```

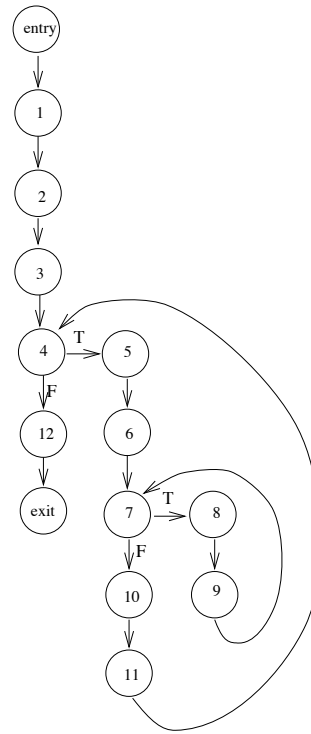


Figure 3: Program segment on the left, with its CFG on the right.

Program Trivial

```

1.  read(n);
2.  if (n < 0) then
3.      write("negative");
4.  else
5.      write("positive");
6.  endif;
7.  switch (n)
8.      case 1:
9.          write("one");
10.         break;
11.     case 2:
12.         write("two");
13.     case 3:
14.         write("three");
15.         break;
16.     default:
17.         write("other");
18. endswitch;
19. end Trivial

```

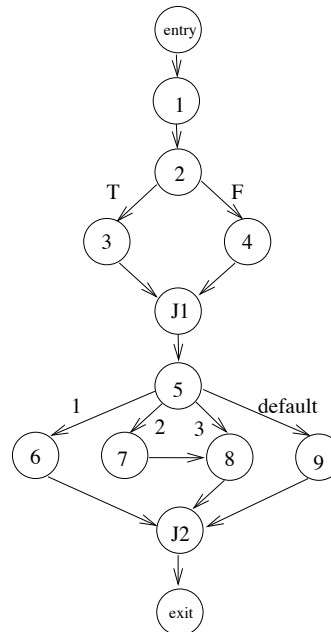


Figure 4: Program segment on the left, with its CFG on the right.

3 Data Flow Information

We can classify each reference to a variable in a program as a definition or a use. A *definition* of a variable occurs whenever a variable gets a value. For example, in Figure 3, there is a definition of variable `n` in statement 1 due to the input statement, and there is a definition of variable `sum` in statement 5 because of the assignment statement. A *use* of a variable occurs whenever the value of the variable is fetched. Uses are further classified as either computation uses (c-uses) or predicate uses (p-use) [11]. A *computation use* occurs whenever a variable either directly affects the computation being performed or is output. A *predicate-use* directly affects the control flow through the program but only indirectly affects a computation. In Figure 3, there are c-uses of `sum` and `j` in statement 8 because their values directly affect the value of `sum` computed in that statement. Likewise, there are c-uses of `sum` and `i` in statement 10 because their values directly affect the program output at that statement. On the other hand, the uses of `j` and `i` in statement 7 directly affect the flow of control from that statement and thus, are p-uses.

By inspecting each statement (basic block) in a program, we can identify the definitions and uses in that statement (basic block). This identification is called *local* data flow analysis. Although local data flow information can be useful for activities such as local optimization, many important compiler and software engineering tasks require information about the flow of data *across* statements (basic blocks). Data flow analysis that computes information across statements (basic blocks) is called *global* data flow analysis, or equivalently, *intraprocedural* data flow analysis.

One well known data flow analysis problem involves the computation of *reaching definitions*. Given program P with CFG G . We say that a definition d *reaches* a point p in G if there is a path in G from the point immediately following d to p , such that d is not “killed” along that path. A definition of a variable v is *killed* at some node n if there is a definition of v at the statement that corresponds to n . For example, in Figure 3, there is a definition of `sum` in statement 5. One way to reach statement 9 is along the subpath 5, 6, 7, 8, 9. However, along this subpath, the definition of `sum` in statement 5 is killed by the definition of `sum` in statement 8. Thus, the definition of `sum` in statement 5 does not reach statement 9; the definition of `sum` in statement 8, however, does reach statement 9.

One way to find the points in a program that definitions reach is to consider each definition in the program individually, and “propagate” it along all paths from the definition until either the definition is killed or an exit from the program is reached. However, this approach may be inefficient because it may require many traversals of the graph. A more efficient approach, and the approach that is most commonly used, is *iterative dataflow analysis*.

Iterative dataflow analysis.

Iterative dataflow analysis², computes four sets of information about each node in the CFG: these sets are entitled **GEN**, **KILL**, **IN**, and **OUT**. Remember that a node may correspond to a statement or a basic block.

- The **GEN** set for a node is the set of definitions in the node that reach the point immediately after the node. For example, the **GEN** set for node 8 in Figure 3 is $\{8:\text{sum}\}$, while the **GEN** set for node 7 in the figure is $\{\}$ (the empty set).
- The **KILL** set for a node is the set of definitions in the program that are killed if they reach the entry to the node. For example, the **KILL** set for node 8 in Figure 3 is $\{3:\text{sum}, 5:\text{sum}, 8:\text{sum}\}$, while the **KILL** set

²See Reference [2] for additional discussion of data flow analysis.

algorithm ReachingDefs

Input. A flow graph for which $KILL[n]$ and $GEN[n]$ have been computed for each node n .

Output. $IN[n]$ and $OUT[n]$ for each node n .

Method. Use an iterative approach, starting with $IN[n] = \{\}$ for all n , and converging to the desired values of IN and OUT . Iterate until the sets do not change; variable **change** records whether a change has occurred on any pass. Here is the algorithm:

```
1. for each node  $n$  do  $OUT[n] = GEN[n]$  endfor
2.  $change = true$ 
3. while  $change$  do
4.    $change = false$ 
5.   for each node  $n$  do
6.      $IN[n] = \cup OUT[P]$ , where  $P$  is an immediate predecessor of  $n$ 
7.      $OLDOUT = OUT[n]$ 
8.      $OUT[n] = GEN[n] \cup (IN[n] - KILL[n])$ 
9.     if  $OUT[n] \neq OLDOUT$  then  $change = true$  endif
10.  endfor
11. endwhile
```

Figure 5: Algorithm for computing IN and OUT sets for reaching definitions.

for node 7 in the figure is $\{\}$ (the empty set). The reason for requiring this set to be a set of definitions *in the program*, rather than in the node, may not initially be clear, but should become more obvious after you see how we use this set in our algorithm for computing reaching definitions.

- The IN set for a node is the set of definitions in the program that reach the point immediately before the node. Notice that the IN set for a node N consists of all definitions that reach the ends of nodes that immediately precede N : that is, $IN[n] = \cup$ (over all P that are predecessors of n) $OUT[P]$.
- The OUT set for a node is the set of definitions in the program that reach the point immediately following the node. Notice that the OUT set for a node N consists of all definitions that either (i) are generated in N , or (ii) reach the entry to N but are not killed within N : that is, $OUT[n] = GEN[n] \cup (IN[n] - KILL[n])$.

Figure 5 gives algorithm **ReachingDefs**, that computes IN and OUT sets for reaching definitions. The algorithm inputs a CFG, with GEN and $KILL$ sets for the nodes in that graph. The algorithm initializes the OUT sets for each node to the GEN sets for those nodes, and then begins a loop (line 3) that repeats as long as sets continue to change. In each iteration of the loop, the algorithm considers each node n (line 5), calculating the IN and OUT sets for n . Line 9 detects whether some OUT set has changed and records this information, thus controlling the iteration of the loop at line (3). When the algorithm completes, the IN sets contain the definitions that reach nodes.

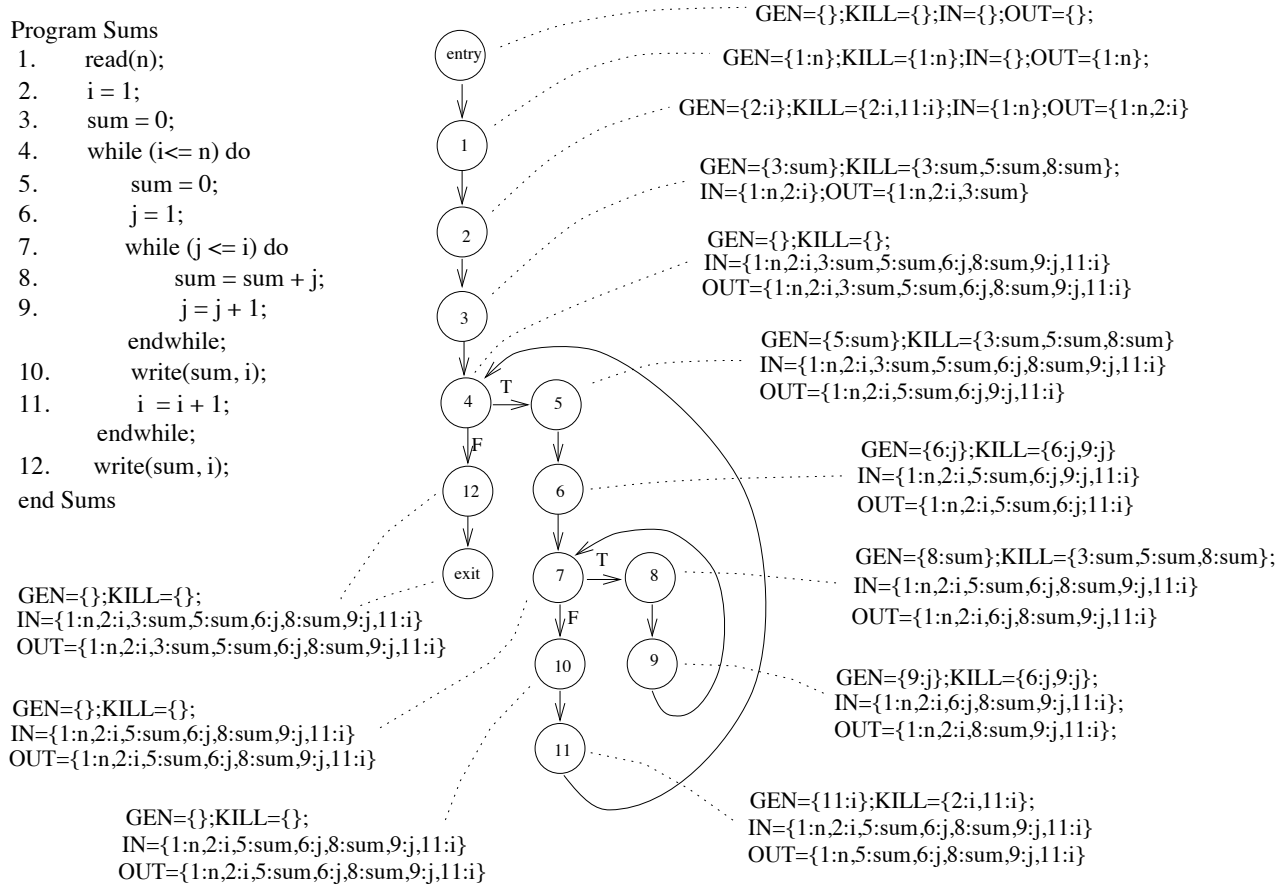


Figure 6: Program segment on the left, with its CFG in the center, and the IN, OUT, GEN, and KILL sets that algorithm `ReachingDefs` computes for the program.

Intuitively, algorithm `ReachingDefs` propagates definitions as far as they will go without being killed – in a sense, simulating all possible executions of the program. In an implementation of this algorithm, we can represent sets of definitions as bit vectors, and perform set operations using logical operations on these bit vectors.

The reason for having KILL sets contain all definitions in the program can now be explained. KILL sets are computed during initialization; at that point we do not know which definitions, in the program, might propagate around the graph and reach the entry to the node. However, we know that any definition of a variable defined in the node will be killed if it reaches the entry to the node; so we create KILL as the set of definitions that, if they reach the node entry, will be killed. During propagation, only definitions that actually reach the entry are affected by the KILL sets, and not propagated through to OUT sets.

Figure 6 shows the IN, OUT, GEN, and KILL sets that algorithm `ReachingDefs` computes for the example program originally given in Figure 3.

There are many other problems that can be solved by data flow analysis, such as calculations of *reachable uses*, *available expressions*, and *live variables*. Reference [2] describes some of these other problems.

Definition-use pairs and data dependence graphs.

A *definition-use pair* for variable v is an ordered pair (D,U) — where D is a statement that contains a definition of v and U is a statement that contains a use of v — such that there is a subpath in the CFG from D to U along which D is not killed. Given reaching definitions information, we can easily compute definition-use pairs: algorithm `ComputeDefUsePairs` of Figure 7 gives an algorithm (not the most efficient).

algorithm `ComputeDefUsePairs`

Input. A flow graph for which the IN sets for reaching definitions have been computed for each node n .

Output. DUPairs: a set of definition-use pairs.

Method. Visit each node in the control flow graph. For each node, use upwards exposed uses and reaching definitions to form definition-use pairs. Here is the algorithm:

```
1. DUPairs = {}
2. for each node  $n$  do
3.     for each upwards exposed use  $U$  in  $n$  do
4.         for each reaching definition  $D$  in  $IN[n]$  do
5.             if  $D$  is a definition of  $v$  and  $U$  is a use of  $v$  then  $DUPairs = DUPairs \cup (D,U)$  endif
6.         endfor
7.     endfor
8. endfor
```

Figure 7: Algorithm for computing definition-use pairs.

Line 3 of the algorithm refers to “upwards exposed” uses. An *upwards exposed use* in node n is a use of some variable v that can be reached by a definition of v that reaches the node. If n contains a statement that defines v , and then after that statement, contains a statement that uses v , the use of v is not upwards exposed: it cannot be reached by a definition that reaches the node. Table 1 lists the definition-use pairs for the example CFG of Figure 3, organized by the node in which the use is located.

Definition-use pairs represent data interactions between statements in a program. If (D,U) is a definition-use pair, then the computation at U is dependent upon data computed at D . We call such a dependence a *data dependence*. There are actually several types of data dependencies: data dependencies of the type we have just described are also called *flow dependencies*. Other types of dependence include *output dependence* and *anti-dependence*. We will consider only flow dependence.

We can represent data dependencies graphically, using a data dependence graph (DDG). A DDG contains node that represent locations of definitions and uses, and edges that represent data dependencies between the nodes. Alternatively, we can add data dependence edges to a CFG. To do this, for each definition-use pair (D,U) that exists in the program that corresponds to the graph, we add edge (D,U) to the graph. Figure 8 partially shows such a graph for program `Sums` (the figure shows only the flow dependence edges that are related to variable `i`). Flow dependence edges are depicted by dotted lines, and annotated by the name of the variable that creates the dependence.

Node	Definition-Use Pairs
entry	(none)
1	(none)
2	(none)
3	(none)
4	[1:n,4:n],[2:i,4:i],[11:i,4:i]
5	(none)
6	(none)
7	[6:j,7:j],[9:j,7:j],[2:i,7:i],[11:i,7:i]
8	[5:sum,8:sum],[8:sum,8:sum],[6:j,8:j],[9:j,8:j]
9	[6:j,9:j],[9:j,9:j]
10	[2:i,10:i],[11:i,10:i],[5:sum,10:sum],[8:sum,10:sum]
11	[2:i,11:i]
12	[2:i,12:i],[11:i,12:i],[3:sum,12:sum],[5:sum,12:sum],[8:sum,12:sum]
exit	(none)

Table 1: Definition-use pairs for program Sums.

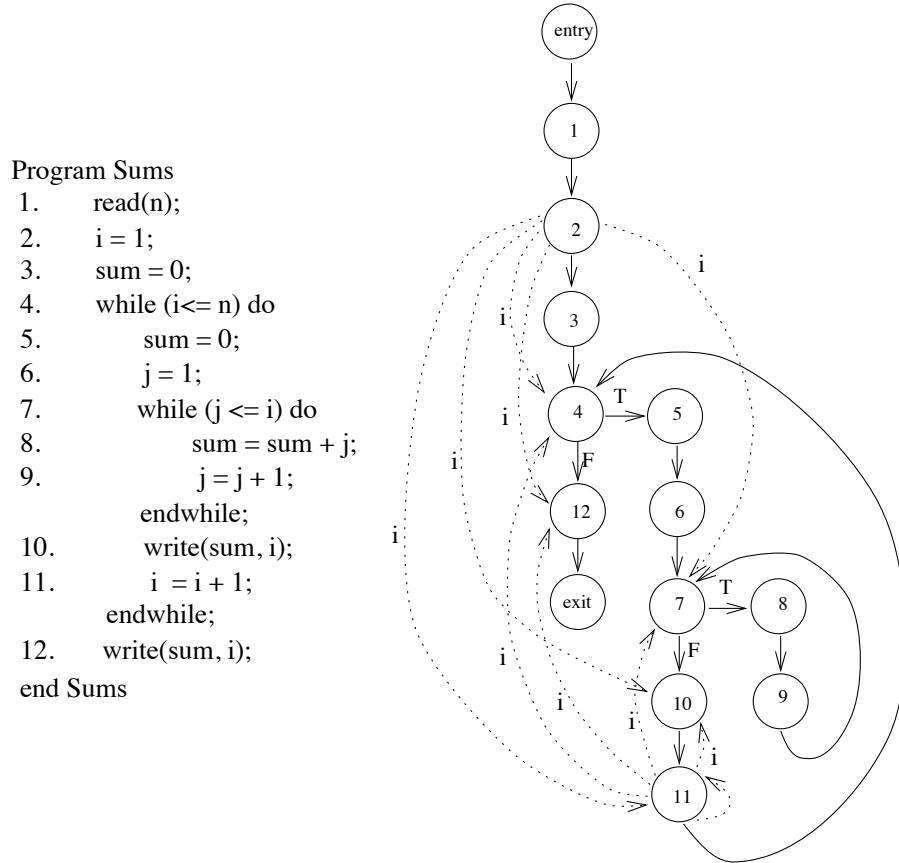


Figure 8: Control flow graph for Sums, with flow dependence edges for i added (dotted lines).

4 Program Paths

A *path* (or *subpath*) in a CFG is a finite sequence of nodes (n_1, n_2, \dots, n_k) such that for $i = 1, 2, \dots, k-1$, there is an edge from n_i to n_{i+1} . A *complete path* is one whose initial node is the entry node and whose final node is the exit node. Examples of paths in Figure 3 are (entry,1,2,3,4,12,exit), which is a complete path, and (6,7,8,9,7,8,9,7,10).

A path (i, n_1, \dots, n_m, j) is *definition-clear* (def-clear) with respect to a variable v from nodes i to j if there is no definition of v in any of the n_i on the path. Similarly, a path $(i, n_1, \dots, n_m, j, k)$ is *definition-clear* with respect to a variable v from node i to edge (j, k) if there is no definition of v in any of the n_i on the path. In Figure 3, (9, 7, 8) is a def-clear path with respect to variable j from node 9 to node 8, and (3,4,5,6,7,8) is a def-clear path with respect to variable i from node 3 to edge (7, 8). On the other hand, (5,6,7,8,9,7,10) is not a def-clear path with respect to `sum` from node 5 to node 10 because `sum` is defined in node 8.

For a definition of variable v in n_1 and a c-use of variable v in n_j , a path (n_1, \dots, n_j) that is def-clear with respect to v in which all nodes except possibly n_1 and n_j are distinct is a *du-path* with respect to v . For a definition of variable v in n_1 and a p-use of variable v in (n_j, n_k) , a path (n_1, \dots, n_j, n_k) that is def-clear with respect to v , in which all nodes n_1, \dots, n_j are distinct is a *du path* with respect to v . Examples of du paths in `Sums` are (3,4,12) for `sum` and (11,4,5,6,7,10) for `i`.

A path through the CFG which, due to conditional statements, can not be executed by any input to the program, is called *infeasible*. Consider path (entry,1,2,3,4,5,6,7,10) through the CFG of Figure 3. For node 5 to execute, `n` must be at least 1; assume that this is the case. When node (statement) 2 executes, `i` is assigned the value 1 and the *while* loop at node (statement) 4 is entered. When the *while* loop at node (statement) 4 is entered, `j` is always assigned the value 1. Thus, when node (statement) 7 is reached during the first iteration of the outer *while* loop, the inner *while* loop is always executed. Since no input can cause path (entry,1,2,3,4,5,6,7,10) to be executed, it is infeasible.

5 Postdominator Trees

A postdominator tree (PDT) describes the postdominance relationship between nodes in a CFG (statements in a program). A node D in CFG G is *postdominated* by a node W in G if and only if every directed path from D to exit (not including D) contains W . A *postdominator tree* is a tree in which the initial node is the exit node, and each node postdominates only its descendants in the tree. Figure 9 shows the CFG and PDT for `Sums`.

We can also define a dominator relationship: a node D in CFG G *dominates* a node W in G if and only if every directed path from entry to W (not including W) contains D . A *dominator tree* is a tree in which the initial node is the entry node, and each node dominates only its descendants in the tree.

Figure 10 gives an algorithm, `ComputeDom`, for computing dominators for a control flow graph G , based on the algorithm presented in [2]. A key to this algorithm is step 3, where, for each node n except the entry node, we initialize the set of dominators to *the set of all nodes in G* . We then iterate through the nodes (except the entry node), and for each node n , at step 3, we use the intersection operator to *reduce* the set of nodes listed as dominating n to those that actually dominate predecessors of n . Thus, we start with an overestimate of the dominators and reduce the sets to get the actual set of dominators. It will help if you try this algorithm on the CFG for `Sums`.

To compute postdominators for a control flow graph G , obtain the *reverse control flow graph* for G by reversing the directions on all edges, and use `ComputeDom` on that reverse control flow graph.

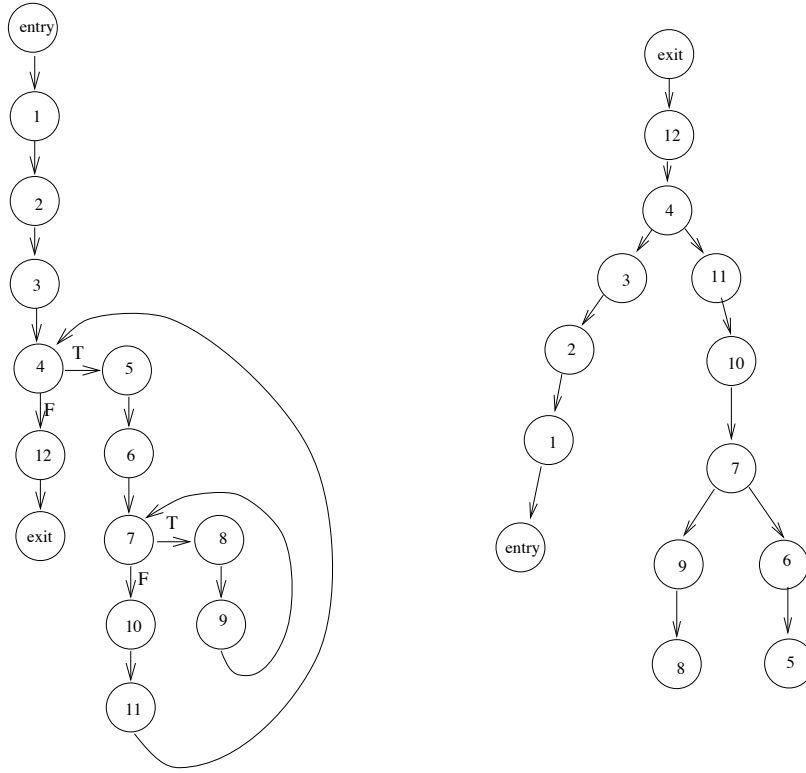


Figure 9: CFG (left) and PDT (right) for **Sums**.

algorithm ComputeDom

Input. A control flow graph G with set of nodes N and initial node n_0 .

Output. $D(n)$, the set of nodes that dominate n , for each node n in G

Method. Use an iterative approach similar to the data flow analysis algorithm **ReachingDefs** given in Figure 5. Here is the algorithm.

1. $D(n_0) = \{n_0\}$
2. **for** each node n in $N - \{n_0\}$ **do** $D(n) = N$
3. **while** changes to any $D(n)$ occur **do**
4. **for** n in $N - \{n_0\}$ **do**
5. $D(n) = \{n\} \cup (\cap D(p)$ for all immediate predecessors p of n
6. **endfor**
7. **endwhile**

Figure 10: Algorithm for computing dominators.

algorithm BuildDtree

Input. A set of nodes N for CFG G , with n_0 the entry node for G , and $D(n)$, the set of nodes that dominate n , for each node n in N .

Output. Dominator tree DT for G .

Method. Here is the algorithm:

1. let n_0 be the root of DT ;
2. put n_0 on queue Q ;
3. **for** each node n in N **do** $D(n) = D(n) - n$ **enddo**;
4. **while** Q is not empty **do**
5. $m =$ the next node on Q (remove it from Q);
6. **for** each node n in N such that $D(n)$ is nonempty **do**
7. **if** $D(n)$ contains m
8. $D(n) = D(n) - m$;
9. **if** $D(n)$ is now empty
10. add n to DT as a child of m ;
11. add n to Q ;
12. **endif**
13. **endif**
14. **endfor**
15. **endwhile**

Figure 11: Algorithm for building a dominator tree.

Figure 11 gives algorithm **BuildDtree**, an algorithm for computing a dominator tree. Applying this algorithm to a set of postdominators yields a postdominator tree.

6 Control Dependence Graphs

Another useful graph is a *control dependence graph* (CDG).³ A CDG encodes *control dependencies*. Assume that nodes do not postdominate themselves. Let X and Y be nodes in CFG G . Y is *control dependent* on X iff (1) there exists a directed path P from X to Y with any Z in P (excluding X and Y) postdominated by Y and (2) X is not postdominated by Y . If Y is control dependent on X then X must have two exits where one of the exits always causes Y to be reached, and the other exit may result in Y not being reached.

The nodes in a CDG represent statements, or regions of code that have common control dependencies. We can construct a CDG using algorithm **GetCDG**, which is shown in Figure 12. To illustrate, Figure 13 shows

³CDGs are treated more formally in [3].

algorithm GetCDG

Input. The control flow graph, CFG, for a program

Output. The CDG for the program.

Method.

1. Augment the CFG with a *start* node, and a “T” edge (start, entry) and a “F” edge (start, exit); call this augmented CFG ACFG
2. Construct the *postdominator tree*, PDT, for ACFG.
3. Determine the *control dependencies* for the program, using the following steps.
 - (a) Find S, a set of edges (A,B) in ACFG such that B is not an ancestor of A in PDT.
 - (b) For each edge (A,B) in S, find L, the least common ancestor of A and B in PDT.
 - (c) Consider each edge (A,B) in S and its corresponding L. Traverse backwards in PDT from B to L, marking each node visited; mark L only if L = A.
 - (d) Statements representing all marked nodes are control dependent on A with the label that is on edge (A,B).
4. Optionally, add region nodes to summarize common control dependencies. (We discuss this step later.)

Figure 12: Algorithm to construct a CDG.

the CDG for `Sums`, without region nodes. Node numbers in the CDG correspond to statement numbers in `Sums`. A CDG contains several types of nodes. Statement nodes, shown as circles in Figure 13, represent simple statements in the program. Predicate nodes, from which labeled edges originate, are represented as rounded boxes.

An alternative form of CDG adds an additional node type called *region* nodes. Region nodes summarize the control dependencies for statements in the region. Each region node summarizes a unique set of control dependence conditions, that contains dependencies that must hold in order for the statements associated with children nodes of the region node to be executed. Figure 14 gives a version of the CDG from Figure 13 to which region nodes, represented as 6-side figures (someone, what’s the word for that?), have been added. Statement nodes 5, 6, 10, and 11 and predicate node 7 are all control dependent upon predicate 4 being true: region node R2 summarizes this dependence. However, predicate node 7 is also control dependent upon itself being true, so region node R3 summarizes dependence on 4-true *or* 7-true. For historical reasons, the start node and edge (start,R0) are usually omitted from CDGs that have region nodes, making R0 serve as the entry node. The process of adding region nodes to graphs is complex to describe: see [3] for details.

7 Program Dependence Graphs

Another important program representation is the *program dependence graph* (PDG), which represents both control dependencies and data dependencies for a program [3]. A PDG consists of a control dependence graph (called a *control dependence subgraph* when part of a PDG), to which data dependence edges (which, together with the PDG nodes, form a *data dependence subgraph*) have been added.

Program Sums

```

1.  read(n);
2.  i = 1;
3.  sum = 0;
4.  while (i <= n) do
5.      sum = 0;
6.      j = 1;
7.      while (j <= i) do
8.          sum = sum + j;
9.          j = j + 1;
10.         endwhile;
11.         write(sum, i);
12.         i = i + 1;
13.     endwhile;
14.     write(sum, i);
15. end Sums

```

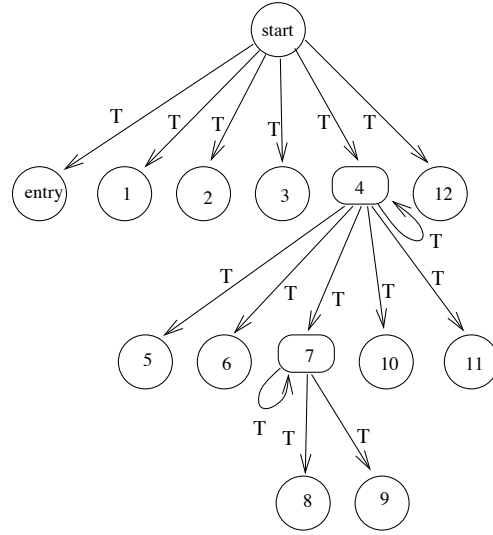


Figure 13: Program Sums on the left and its CDG without region nodes on the right. Circles represent statements in the program, and rounded rectangles represent predicates.

Program Sums

```

1.  read(n);
2.  i = 1;
3.  sum = 0;
4.  while (i <= n) do
5.      sum = 0;
6.      j = 1;
7.      while (j <= i) do
8.          sum = sum + j;
9.          j = j + 1;
10.         endwhile;
11.         write(sum, i);
12.         i = i + 1;
13.     endwhile;
14.     write(sum, i);
15. end Sums

```

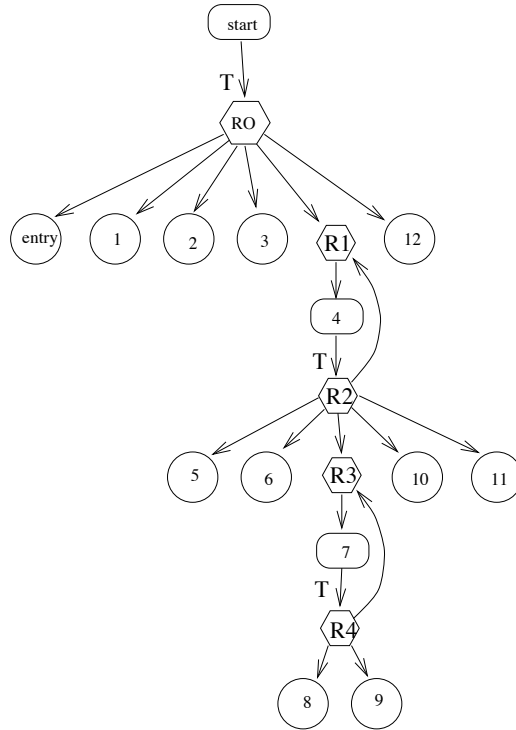


Figure 14: Program Sums on the left and its CDG with region nodes on the right.

PDG node	control dependence		data dependence	
	successors	predecessors	successors	predecessors
entry	none	R0	none	none
1	none	R0	4	none
2	none	R0	4,7,10,11,12	none
3	none	R0	12	none
4	R2	R1	none	1,2,11
5	none	R1	8,10,12	none
6	none	R1	7,8,9	none
7	R4	R3	none	2,6,9,11
8	none	R4	8,10,12	5,6,8,9
9	none	R4	7,8,9	6,9
10	none	R2	none	2,5,8,11
11	none	R2	4,7,10,11,12	2,11
12	none	R0	none	2,3,5,8,11
exit	none	R0	none	none
R0	entry,1,2,3,R1,12,exit	none	none	none
R1	4	R0,R2	none	none
R2	3,6,R3,10,11,R1	4	none	none
R3	7	R2,R4	none	none
R4	8,9,R3	7	none	none

Table 2: PDG nodes corresponding to the PDG for program **Sums** of Figure 3, and the control dependence and data dependence edges.

8 Slicing

The concept of a program slice was originally developed by Weiser [12] for debugging. A *slice* of a program with respect to a program point P and set of program variables V consists of all statements and predicates in the program that may affect the values of variables in V at P . Weiser used the CFG and data flow analysis algorithms for slicing. Subsequent research [3, 10] presents techniques for computing slices using the PDG. We illustrate slicing using the PDG.

Given a PDG that contains both a control dependence subgraph and a data dependence subgraph, a slice at a program point P with respect to a variable v can be obtained using a backward walk from P that includes all nodes in the PDG reachable from P . To illustrate the concept of slicing, consider program **Sums** given in Figure 3. For ease in traversing the PDG, Figure 8 lists the PDG edges for **Sums**.

Suppose we want to determine all statements in **Sums** that affect the value of j in statement (node) 9. We initially add node 9 to a **worklist** used to record the nodes that remain to be visited and initialize **slice** to empty. Then, we remove a node N from **worklist**, mark it as visited, and add N 's unvisited control dependence and data dependence predecessors to **worklist** and **slice**; we repeat this step until **worklist** is empty. The resulting **slice** contains statements 1, 2, 4, 6, 7, 9, and 11, which are all statements in **Sums** that affect the value of j in statement 9.

There have been several useful extensions to slicing algorithms. One problem with the static backward slice that was developed by Weiser is that it contains *all* statements that may affect a given statement

during *any* program execution. A refinement of this static slice that eliminates the problem of additional statements is a dynamic slice [1, 5]. Whereas a static slice is computed for all possible executions of the program, a *dynamic slice* contains only those statements that affect a given statement during one execution of the program. Thus, a dynamic slice is associated with each test case for the program. One technique for computing a dynamic slice marks those nodes and edges in the PDG that are traversed during one execution of a program and then takes a static slice for the statement restricted to the marked nodes.

Another type of slice that is useful is a forward slice[4]. A *forward slice* for a program point P and a variable v consists of all those statements and predicates in the program that might be affected by the value of v at P. A forward slice is computed by taking the transitive closure of the statements directly affected by the value of v at P.

9 Alias Analysis

An *alias* is a name referring to the same memory location as another name, at a given program point. In such a case, that memory locations can be accessed through any of these two names. Alias analyses are concerned with the identification of such aliases within a program. Aliases are introduced by the use of language constructs (e.g., pointers and reference parameters in C, references in Java). As an example, consider the C code provided in Figure 15.

```

1.  void main() {
2.      int x, , y, *p;
3.      read(x);
4.      read(y);
5.      p = &x;
6.      *p = 0;
7.      print(y/x);
8.  }
```

Figure 15: An example of alias.

On line 6, pointer p points to the memory location that contains x . As a consequence, name $*p$ and name x refer to the same entity. In such a case, we say that $*p$ is an alias for x at statement 6. Alias analysis results are conveniently represented in terms of points-to sets (or graphs). Given a point n in the program, and a name of a non-fixed location (i.e., a pointer) p , the *points-to set* for p at n contains all the fixed-locations (i.e., the non-pointer variables) that p may refer to when the execution reaches n . For example, in the code of Figure 15 the points-to set for $*p$ at statement 6 would contain the only element x .

Taking into account aliasing allows for identifying side-effects in a program (i.e., effects due to indirect modifications of the state of the program). In the example of Figure 15, the division by zero occurring at statement 7 is due to the indirect modification of x through $*p$ (i.e., it is due to a side effect of statement 6 related to the presence of aliases.)

A further example is provided in Figure 16. At statement 9, the name $*p$ could refer to any of the two variables x and y , depending on the path followed by the program during its execution (i.e., depending on the execution path). In such a case, the points-to set for $*p$ would include both x and y , as shown in the comment associated with statement 9.

In general, the points-to information is a MAY kind of information. *MAY* information, as opposed to *MUST* information, refers to some statically identified fact which may or may not hold during the execution of the program. Referring to the previous example, name $*p$ at statement 9 may or may not refer to variable x (resp. y), depending on whether the specific execution path traverses statement 6 (resp., 8) or

```

1.  void main() {
2.      int x, y, *p;
3.      x = read();
4.      y = read();
5.      if( x > y )
6.          p=&x;
7.      else
8.          p=&y;
9.      *p = read(); /* points-to set for *p = {x, y} */
10.     print(x);
11. }

```

Figure 16: A further example of alias.

not. Moreover, also when the points-to set contains only one element, the information provided could still be a MAY information, if the algorithm used for alias analysis is not precise enough.⁴ A possible example of that is provided in Figure 17.

```

1.  void main() {
2.      int x, y, *p;
3.      x = read();
4.      y = read();
5.      if( x > y )
6.          p=&x;
7.      *p = read(); /* points-to set for *p = {x} */
8.      print(x);
9.  }

```

Figure 17: A single element, still imprecise, points-to set.

In this example, pointer p can either point to the address of variable x or be undetermined. Therefore, we cannot safely assume that $*p$ always refers to x . Further details on the different ways of computing alias information and the description of a specific algorithm can be found in Reference [6].

10 Data-flow analysis in the presence of structures and pointers

Traditional data-flow analyses, as proposed for simple Pascal-like languages, have to be modified to be applicable to languages as C. In fact, extensive use of pointers and complex structures renders the traditional definition of DUA inadequate.

10.1 Data-flow analysis and pointers

The presence of pointers introduces an additional dimension to the problem of identifying DUAs. To see how the traditional definition of DUA falls short in the presence of pointers, consider again the example of C code provided in Figure 16. The first problem there is how to identify definitions and uses. For example, how do we consider the assignment “ $*p = read();$ ”? According to the traditional definition, the statement contains a definition of $*p$ and no uses. If we take into account aliases this is not enough, since the execution of the statement may also modify the value of either x or y , depending on the execution path. This means that there is a DUA $\langle x, n_9, n_{10} \rangle$ in the program that has to be considered. Unfortunately, we cannot solve the problem by simply adding x and y to the def set for statement 9. Doing so, we would kill any definition

⁴Due to the complexity of alias analysis, algorithms for computing alias information are usually approximated (i.e., imprecise), but conservative (i.e., safe).

of x reaching statement 9, so we would miss the DUA $\langle x, n_3, n_{10} \rangle$. Note that both DUA $\langle x, n_9, n_{10} \rangle$ and DUA $\langle x, n_3, n_{10} \rangle$ are actual DUAs for the program: the former occurs for every input to the program such that $x > y$; the latter occurs for every input to the program such that $x \leq y$.

One of the proposed solutions for this problem is to consider two different kinds of definitions and uses:

DDEF (Definite DEFinition) Traditional definition.

PDEF (Possible DEFinition) Definition of a variable through the dereference of a pointer. Such a definition is in general only possible, being alias information a MAY information.

DUSE (Definite USE) Traditional use.

PUSE (Possible USE) Use of a variable through the dereference of a pointer. Such an use is in general only possible, being alias information a MAY information.

Figure 18 shows the DDEF, PDEF, DUSE, and PUSE sets for the example in Figure 16.

```

1.  void main() {
2.      int x, y, *p; // DDEF={ }   PDEF={ }   DUSE={ }   PUSE={ }
3.      x = read(); // DDEF={x}   PDEF={ }   DUSE={ }   PUSE={ }
4.      y = read(); // DDEF={y}   PDEF={ }   DUSE={ }   PUSE={ }
5.      if( x > y ) // DDEF={ }   PDEF={ }   DUSE={x, y} PUSE={ }
6.          p=&x; // DDEF={p}   PDEF={ }   DUSE={ }   PUSE={ }
7.      else // DDEF={ }   PDEF={ }   DUSE={ }   PUSE={ }
8.          p=&y; // DDEF={p}   PDEF={ }   DUSE={ }   PUSE={ }
9.      *p = read(); // DDEF={*p} PDEF={x, y} DUSE={ }   PUSE={ }
10.     print(x); // DDEF={ }   PDEF={ }   DUSE={x}   PUSE={ }
11. }
```

Figure 18: DDEFs, PDEFs, DUSEs, and PUSEs.

The above sets can be used to tailor traditional data-flow algorithms to the case of languages with pointers. Algorithms are modified as follows: (1) both definite and possible definitions and both definite and possible uses are considered when identifying the definition and the use within a DUA, and (2) only definite definitions are considered to be able to kill other definitions.

An additional concern is whether to consider a dereference of a pointer as a use of the pointer variable. For example, for statement 9 of the previous example, we might want to consider “ $*p = \dots$ ” as both a definite definition of $*p$ (and a possible definitions of $*p$ ’s aliases) and a definite use of p . If so, the data-flow information shown in Figure 18 should be modified to include a definite use of p in statement 9. This would lead to the identification of two additional DUAs: $\langle p, n_6, n_9 \rangle$ and $\langle p, n_8, n_9 \rangle$. Further details on data-flow analysis for languages with pointers can be found in References [7, 8, 9].

10.2 Data-flow analysis and structures

As for pointers, the presence of structures in a program adds an additional dimension to the problem of evaluating DUAs. The main issue here is whether a structure should be considered as a whole or its fields should be distinguished, when performing data-flow analysis. In the former case, a definition (resp., use) of any of the fields of a structure would be treated as a definition (resp., use) of the structure itself. Referring to the example shown in Figure 19, statement 10 would be considered a definition of s , and statement 11 would be considered a use of s . Therefore, the DUA $\langle s, n_{10}, n_{11} \rangle$, which is a spurious DUA, would be identified.

```

1.  struct Bar {
2.      int x;
3.      int y;
4.  };
8.  main() {
9.      struct Bar b;
10.     b.x = read();
11.     print(b.y);
12. }

```

Figure 19: A DUA involving a structure.

This approach allows for a faster analysis, and is still safe, but is less precise than the approach that treats the different fields of a structure as distinguished entities. Applying this latter approach on the last example, we would not have identified the spurious DUA: the definition at statement 10 would have been considered a definition of $s.x$, and the use at statement 11 would have been considered a use of $s.y$.

When pointers and structures are both present at the same time, as it usually happens in C programs, the different issues related to both aspects have to be considered. The example shown in Figure 20 provides some evidence of the complexity of DUAs identification even for a very small-sized C program.

```

1.  struct Bar {
2.      int x;
3.      int y;
4.  };

5.  struct Foo {
6.      struct Bar* p_to_bar_in_foo;
7.  }

8.  main() {
9.      int i, l;
10.     struct Foo* p_to_foo, foo;
11.     struct Bar bar;
12.     l = read();
13.     p_to_foo = &foo;
14.     foo.p_to_bar_in_foo->p_to_y_in_bar = &i;
15.     bar.p_to_y_in_bar = &l;
16.     foo.p_to_bar_in_foo = &bar;
17.     print(*p_to_foo->p_to_bar_in_foo->p_to_y_in_bar);
18. }

```

Figure 20: C code containing pointers and structures.

Consider for example statement 17. What is actually, or even possibly, defined when such statement is executed? And what is used? The answers to those and similar questions highly depend on several factors, among which the most relevant are the precision of the alias information available and the precision/efficiency of the data-flow analysis we want to perform. Further details about the issues related to pointers and structures when program analysis is concerned can be found in References [7, 13].

References

- [1] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation. SIGPLAN Notices*, pages 246–56, June 1990.

- [2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.
- [3] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.
- [4] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.
- [5] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–63, October 1988.
- [6] D. Liang and M. J. Harrold. Efficient points-to analysis for whole-program analysis. In *Proceedings of ESEC/FSE '99*, volume 1687 of *LNCS*, pages 199–215. Springer-Verlag, September 1999.
- [7] A. Orso, D. Liang, S. Sinha, and M. J. Harrold. A framework for understanding data dependences. Technical Report GIT-CC-02-13, College of Computing, Georgia Institute of Technology, March 2002.
- [8] Alessandro Orso, Saurabh Sinha, and Mary Jean Harrold. Effects of pointers on data dependences. In *Proceedings of the 9th International Workshop on Program Comprehension (IWPC 2001)*, pages 39–49, Toronto, Ontario, Canada, May 2001.
- [9] Alessandro Orso, Saurabh Sinha, and Mary Jean Harrold. Incremental slicing based on data-dependence types. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 158–167, Firenze, Italy, November 2001.
- [10] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184, March 1984.
- [11] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, April 1985.
- [12] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.
- [13] Suan Hsi Yong, Susan Horwitz, and Thomas Reps. Pointer analysis for programs with structures and casting. *ACM SIGPLAN Notices*, 34(5):91–103, May 1999.