Chapter 6

Dependence and Data Flow Models

The control flow graph and state machine models introduced in the previous chapter capture one aspect of the dependencies among parts of a program. They explicitly represent control flow but deemphasize transmission of information through program variables. Data flow models provide a complementary view, emphasizing and making explicit relations involving transmission of information.

Models of data flow and dependence in software were originally developed in the field of compiler construction, where they were (and still are) used to detect opportunities for optimization. They also have many applications in software engineering, from testing to refactoring to reverse engineering. In test and analysis, applications range from selecting test cases based on dependence information (as described in Chapter 13) to detecting anomalous patterns that indicate probable programming errors, such as uses of potentially uninitialized values. Moreover, the basic algorithms used to construct data flow models have even wider application and are of particular interest because they can often be quite efficient in time and space.

6.1 Definition-Use Pairs

The most fundamental class of data flow model associates the point in a program where a value is produced (called a "definition") with the points at which the value may be accessed (called a "use"). Associations of definitions and uses fundamentally capture the flow of information through a program, from input to output.

Definitions occur where variables are declared or initialized, assigned values, or received as parameters, and in general at all statements that change the value of one or more variables. Uses occur in expressions, conditional statements, parameter passing, return statements, and in general in all statements whose execution extracts a value from a variable. For example, in the standard greatest common divisor (GCD) algorithm of Figure 6.1, line 1 contains a definition of parameters x and y, line 3 contains a use of variable y, line 6 contains a use of variable tmp and a definition of variable y,

```
1    public int gcd(int x, int y) {        /* A: def x,y   */
2        int tmp;                           /*      def tmp  */
3        while (y != 0) {                   /* B: use y     */
4            tmp = x % y;                   /* C: use x,y, def tmp */
5            x = y;                         /* D: use y, def x  */
6            y = tmp;                       /* E: use tmp, def y */
7        }
8        return x;                          /* F: use x */
9    }
```

Figure 6.1: Java implementation of Euclid's algorithm for calculating the greatest common denominator of two positive integers. The labels A–F are provided to relate statements in the source code to graph nodes in subsequent figures.

and the return in line 8 is a use of variable x.

Each definition-use pair associates a definition of a variable (e.g., the assignment to y in line 6) with a use of the same variable (e.g., the expression y != 0 in line 3). A single definition can be paired with more than one use, and vice versa. For example, the definition of variable y in line 6 is paired with a use in line 3 (in the loop test), as well as additional uses in lines 4 and 5. The definition of x in line 5 is associated with uses in lines 4 and 8.

A definition-use pair is formed only if there is a program path on which the value assigned in the definition can reach the point of use without being overwritten by another value. If there is another assignment to the same value on the path, we say that the first definition is *killed* by the second. For example, the declaration of tmp in line 2 is not paired with the use of tmp in line 6 because the definition at line 2 is killed by the definition at line 4. A *definition-clear* path is a path from definition to use on which the definition is not killed by another definition of the same variable. For example, with reference to the node labels in Figure 6.2, path E,B,C,D is a definition-clear path from the definition of y in line 6 (node E of the control flow graph) to the use of y in line 5 (node D). Path A,B,C,D,E is not a definition-clear path with respect to tmp because of the intervening definition at node C.

Δ kill

Δ definition-clear path

Definition-use pairs record a kind of program dependence, sometimes called direct data dependence. These dependencies can be represented in the form of a graph, with a directed edge for each definition-use pair. The data dependence graph representation of the GCD method is illustrated in Figure 6.3 with nodes that are program statements. Different levels of granularity are possible. For use in testing, nodes are typically basic blocks. Compilers often use a finer-grained data dependence representation, at the level of individual expressions and operations, to detect opportunities for performance-improving transformations.

Δ direct data dependence

The data dependence graph in Figure 6.3 captures only dependence through flow of data. Dependence of the body of the loop on the predicate governing the loop is not represented by data dependence alone. Control dependence can also be represented with a graph, as in Figure 6.5, which shows the control dependencies for the GCD

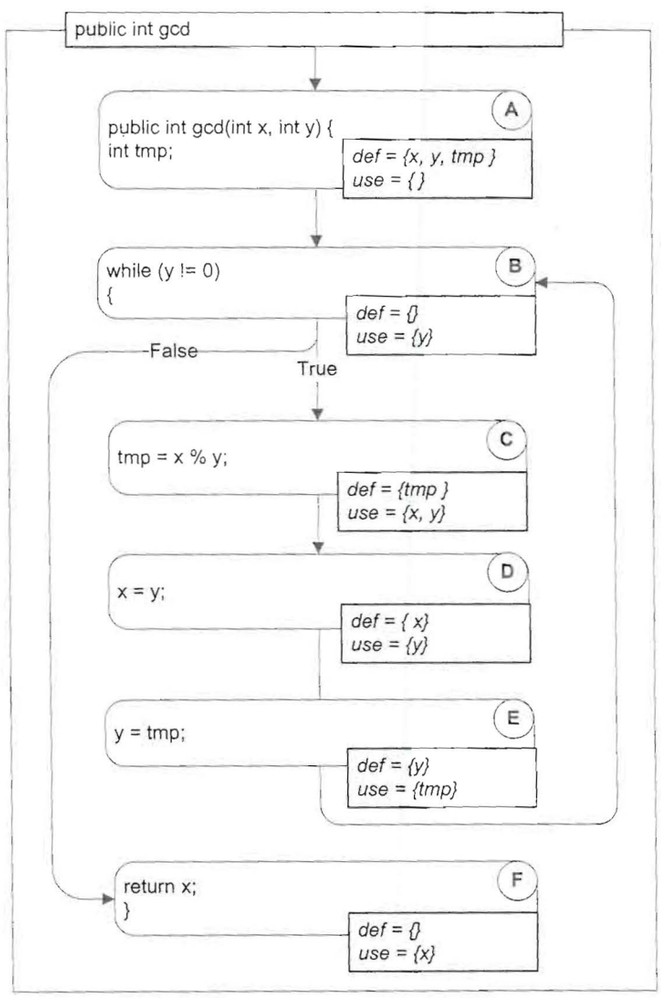

Figure 6.2: Control flow graph of GCD method in Figure 6.1.

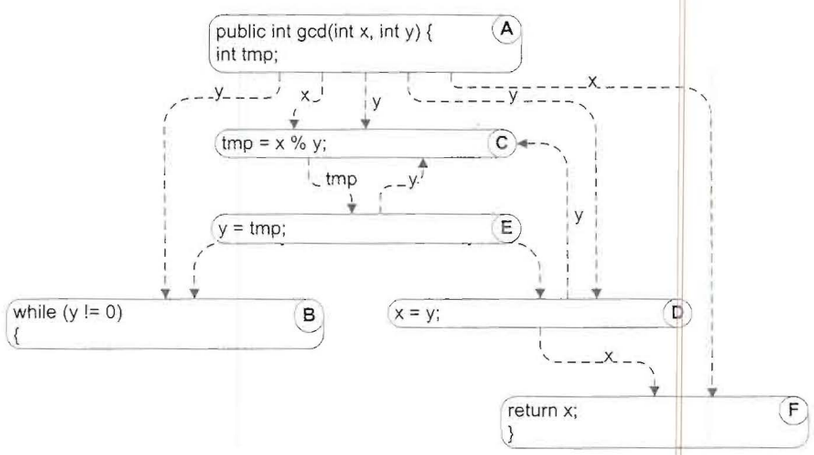

Figure 6.3: Data dependence graph of GCD method in Figure 6.1, with nodes for statements corresponding to the control flow graph in Figure 6.2. Each directed edge represents a direct data dependence, and the edge label indicates the variable that transmits a value from the definition at the head of the edge to the use at the tail of the edge.

method. The control dependence graph shows direct control dependencies, that is, where execution of one statement controls whether another is executed. For example, execution of the body of a loop or if statement depends on the result of a predicate.

Control dependence differs from the sequencing information captured in the control flow graph. The control flow graph imposes a definite order on execution even when two statements are logically independent and could be executed in either order with the same results. If a statement is control- or data-dependent on another, then their order of execution is not arbitrary. Program dependence representations typically include both data dependence and control dependence information in a single graph with the two kinds of information appearing as different kinds of edges among the same set of nodes.

A node in the control flow graph that is reached on every execution path from entry point to exit is control dependent only on the entry point. For any other node N, reached on some but not all execution paths, there is some branch that controls execution of N in the sense that, depending on which way execution proceeds from the branch, execution of N either does or does not become inevitable. It is this notion of control that control dependence captures.

Δ dominator

The notion of dominators in a rooted, directed graph can be used to make this intuitive notion of "controlling decision" precise. Node M dominates node N if every path from the root of the graph to N passes through M. A node will typically have many dominators, but except for the root, there is a unique *immediate dominator* of node N, which is closest to N on any path from the root and which is in turn dominated

Δ immediate dominator

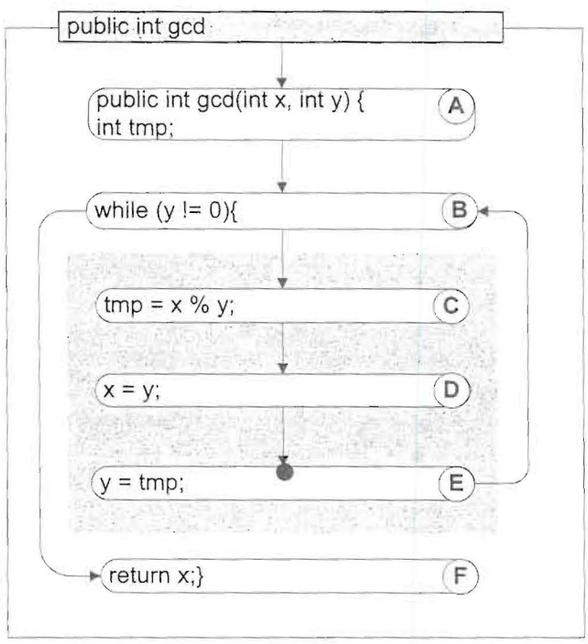

Figure 6.4: Calculating control dependence for node E in the control flow graph of the GCD method. Nodes C, D, and E in the gray region are post-dominated by E; that is, execution of E is inevitable in that region. Node B has successors both within and outside the gray region, so it controls whether E is executed; thus E is control-dependent on B.

by all the other dominators of N. Because each node (except the root) has a unique immediate dominator, the immediate dominator relation forms a tree.

The point at which execution of a node becomes inevitable is related to paths from a node to the end of execution — that is, to dominators that are calculated in the reverse of the control flow graph, using a special "exit" node as the root. Dominators in this direction are called post-dominators, and dominators in the normal direction of execution can be called pre-dominators for clarity.

Δ post-dominator

Δ pre-dominator

We can use post-dominators to give a more precise definition of control dependence. Consider again a node N that is reached on some but not all execution paths. There must be some node C with the following property: C has at least two successors in the control flow graph (i.e., it represents a control flow decision); C is not post-dominated by N (N is not already inevitable when C is reached); and there is a successor of C in the control flow graph that is post-dominated by N. When these conditions are true, we say node N is control-dependent on node C. Figure 6.4 illustrates the control dependence calculation for one node in the GCD example, and Figure 6.5 shows the control dependence relation for the method as a whole.

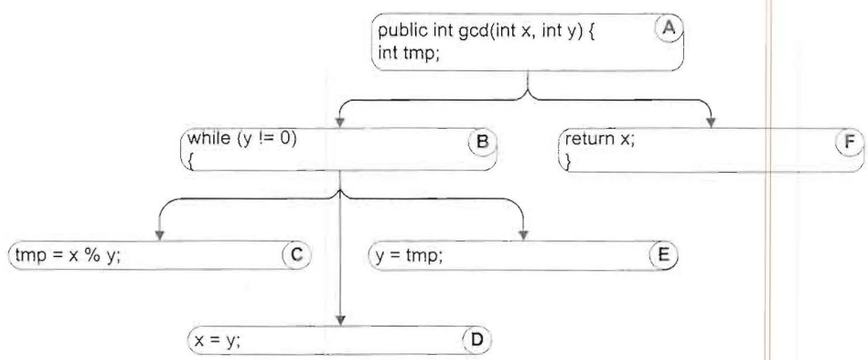

Figure 6.5: Control dependence tree of the GCD method. The loop test and the return statement are reached on every possible execution path, so they are control-dependent only on the entry point. The statements within the loop are control-dependent on the loop test.

6.2 Data Flow Analysis

Δ reaching definition

Definition-use pairs can be defined in terms of paths in the program control flow graph. As we have seen in the former section, there is an association (d,u) between a definition of variable v at d and a use of variable v at u if and only if there is at least one control flow path from d to u with no intervening definition of v. We also say that definition v_d *reaches* u, and that v_d is a *reaching definition* at u. If, on the other hand, a control flow path passes through another definition e of the same variable v, we say that v_e *kills* v_d at that point.

It would be possible to compute definition-use pairs by searching the control flow graph for individual paths of the form described above. However, even if we consider only loop-free paths, the number of paths in a graph can be exponentially larger than the number of nodes and edges. Practical algorithms therefore cannot search every individual path. Instead, they summarize the reaching definitions at a node over all the paths reaching that node.

An efficient algorithm for computing reaching definitions (and several other properties, as we will see below) is based on the way reaching definitions at one node are related to reaching definitions at an adjacent node. Suppose we are calculating the reaching definitions of node n, and there is an edge (p,n) from an immediate predecessor node p. We observe:

- If the predecessor node p can assign a value to variable v, then the definition v_p reaches n. We say the definition v_p is *generated* at p.
- If a definition v_d of variable v reaches a predecessor node p, and if v is not redefined at that node (in which case we say the v_d is *killed* at that point), then the definition is propagated on from p to n.

These observations can be stated in the form of an equation describing sets of reaching definitions. For example, reaching definitions at node E in Figure 6.2 are those at node D, except that D adds a definition of y and replaces (kills) an earlier definition of y:

$$Reach(E) = (Reach(D) \setminus \{x_A\}) \cup \{x_D\}$$

This rule can be broken down into two parts to make it a little more intuitive and more efficient to implement. The first part describes how node E receives values from its predecessor D, and the second describes how it modifies those values for its successors:

$$\begin{aligned} Reach(E) &= ReachOut(D) \\ ReachOut(D) &= (Reach(D) \setminus \{x_A\}) \cup \{x_D\} \end{aligned}$$

In this form, we can easily express what should happen at the head of the while loop (node B in Figure 6.2), where values may be transmitted both from the beginning of the procedure (node A) and through the end of the body of the loop (node E). The beginning of the procedure (node A) is treated as an initial definition of parameters and local variables. (If a local variable is declared but not initialized, it is treated as a definition to the special value "uninitialized.")

$$\begin{aligned} Reach(B) &= ReachOut(A) \cup ReachOut(E) \\ ReachOut(A) &= gen(A) = \{x_A, y_A, tmp_A\} \\ ReachOut(E) &= (ReachIn(E) \setminus \{y_A\}) \cup \{y_E\} \end{aligned}$$

In general, for any node n with predecessors $pred(n)$,

$$\begin{aligned} Reach(n) &= \bigcup_{m \in pred(n)} ReachOut(m) \\ ReachOut(n) &= (ReachIn(n) \setminus kill(n)) \cup gen(n) \end{aligned}$$

Remarkably, the reaching definitions can be calculated simply and efficiently, first initializing the reaching definitions at each node in the control flow graph to the empty set, and then applying these equations repeatedly until the results stabilize. The algorithm is given as pseudocode in Figure 6.6.

Algorithm *Reaching definitions*

Input: A control flow graph $G = (\text{nodes}, \text{edges})$
$\text{pred}(n) = \{m \in \text{nodes} \mid (m,n) \in \text{edges}\}$
$\text{succ}(m) = \{n \in \text{nodes} \mid (m,n) \in \text{edges}\}$
$\text{gen}(n) = \{v_n\}$ if variable v is defined at n, otherwise $\{\}$
$\text{kill}(n) =$ all other definitions of v if v is defined at n, otherwise $\{\}$

Output: $\text{Reach}(n) =$ the reaching definitions at node n

```
for n ∈ nodes loop
    ReachOut(n) = {} ;
end loop;
workList = nodes ;
while (workList ≠ {}) loop
    // Take a node from worklist (e.g., pop from stack or queue)
    n = any node in workList ;
    workList = workList \ {n} ;

    oldVal = ReachOut(n) ;

    // Apply flow equations, propagating values from predecessors
    Reach(n) = ∪_{m∈pred(n)} ReachOut(m);
    ReachOut(n) = (Reach(n) \ kill(n)) ∪ gen(n) ;
    if ( ReachOut(n) ≠ oldVal ) then
        // Propagate changed value to successor nodes
        workList = workList ∪ succ(n)
    end if;
end loop;
```

Figure 6.6: An iterative work-list algorithm to compute reaching definitions by applying each flow equation until the solution stabilizes.

6.3 Classic Analyses: Live and Avail

Reaching definition is a classic data flow analysis adapted from compiler construction to applications in software testing and analysis. Other classical data flow analyses from compiler construction can likewise be adapted. Moreover, they follow a common pattern that can be used to devise a wide variety of additional analyses.

Available expressions is another classical data flow analysis, used in compiler construction to determine when the value of a subexpression can be saved and reused rather than recomputed. This is permissible when the value of the subexpression remains unchanged regardless of the execution path from the first computation to the second.

Available expressions can be defined in terms of paths in the control flow graph. An expression is *available* at a point if, for all paths through the control flow graph from procedure entry to that point, the expression has been computed and not subsequently modified. We say an expression is *generated* (becomes available) where it is computed and is *killed* (ceases to be available) when the value of any part of it changes (e.g., when a new value is assigned to a variable in the expression).

As with reaching definitions, we can obtain an efficient analysis by describing the relation between the available expressions that reach a node in the control flow graph and those at adjacent nodes. The expressions that become available at each node (the *gen* set) and the expressions that change and cease to be available (the *kill* set) can be computed simply, without consideration of control flow. Their propagation to a node from its predecessors is described by a pair of set equations:

$$Avail(n) = \bigcap_{m \in pred(n)} AvailOut(m)$$

$$AvailOut(n) = (Avail(n) \setminus kill(n)) \cup Gen(n)$$

The similarity to the set equations for reaching definitions is striking. Both propagate sets of values along the control flow graph in the direction of program execution (they are *forward* analyses), and both combine sets propagated along different control flow paths. However, reaching definitions combines propagated sets using set union, since a definition can reach a use along *any* execution path. Available expressions combines propagated sets using set intersection, since an expression is considered available at a node only if it reaches that node along *all* possible execution paths. Thus we say that, while reaching definitions is a *forward, any-path* analysis, available expressions is a *forward, all-paths* analysis. A work-list algorithm to implement available expressions analysis is nearly identical to that for reaching definitions, except for initialization and the flow equations, as shown in Figure 6.7.

forward analysis

any-path analysis

all-paths analysis

Applications of a forward, all-paths analysis extend beyond the common subexpression detection for which the Avail algorithm was originally developed. We can think of available expressions as tokens that are propagated from where they are generated through the control flow graph to points where they might be used. We obtain different analyses by choosing tokens that represent some other property that becomes true (is generated) at some points, may become false (be killed) at some other points, and is

Algorithm *Available expressions*

Input: A control flow graph $G = (\text{nodes}, \text{edges})$, with a distinguished root node *start*.
$\text{pred}(n) = \{m \in \text{nodes} \mid (m,n) \in \text{edges}\}$
$\text{succ}(m) = \{n \in \text{nodes} \mid (m,n) \in \text{edges}\}$
$\text{gen}(n)$ = all expressions e computed at node n
$\text{kill}(n)$ = expressions e computed anywhere, whose value is changed at n;
$\text{kill}(start)$ is the set of all e.

Output: $\text{Avail}(n)$ = the available expressions at node n

```
for n ∈ nodes loop
    AvailOut(n) = set of all e defined anywhere ;
end loop;
workList = nodes ;
while (workList ≠ {}) loop
    // Take a node from worklist (e.g., pop from stack or queue)
    n = any node in workList ;
    workList = workList \ {n} ;
    oldVal = AvailOut(n) ;
    // Apply flow equations, propagating values from predecessors
    Avail(n) = ⋂_{m∈pred(n)} AvailOut(m);
    AvailOut(n) = (Avail(n) \ kill(n)) ∪ gen(n) ;
    if ( AvailOut(n) ≠ oldVal ) then
        // Propagate changes to successors
        workList = workList ∪ succ(n)
    end if;
end loop;
```

Figure 6.7: An iterative work-list algorithm for computing available expressions.

```
1   /** A trivial method with a potentially uninitialized variable.
2     * Java compilers reject the program. The compiler uses
3     * data flow analysis to determine that there is a potential
4     * (syntactic) execution path on which k is used before it
5     * has been assigned an initial value.
6     */
7   static void questionable() {
8       int k;
9       for (int i=0; i < 10; ++i) {
10          if (someCondition(i)) {
11              k = 0;
12          } else {
13              k += i;
14          }
15      }
16      System.out.println(k);
17  }
18 }
```

Figure 6.8: Function questionable *(repeated from Chapter 3) has a potentially uninitialized variable, which the Java compiler can detect using data flow analysis.*

evaluated (used) at certain points in the graph. By associating appropriate sets of tokens in gen and kill sets for a node, we can evaluate other properties that fit the pattern

> "G occurs on all execution paths leading to U, and there is no intervening occurrence of K between the last occurrence of G and U."

G, K, and U can be any events we care to check, so long as we can mark their occurrences in a control flow graph.

An example problem of this kind is variable initialization. We noted in Chapter 3 that Java requires a variable to be initialized before use on all execution paths. The analysis that enforces this rule is an instance of Avail. The tokens propagated through the control flow graph record which variables have been assigned initial values. Since there is no way to "uninitialize" a variable in Java, the kill sets are empty. Figure 6.8 repeats the source code of an example program from Chapter 3. The corresponding control flow graph is shown with definitions and uses in Figure 6.9 and annotated with gen and kill sets for the initialized variable check in Figure 6.10.

Reaching definitions and available expressions are forward analyses; that is, they propagate values in the direction of program execution. Given a control flow graph model, it is just as easy to propagate values in the opposite direction, backward from nodes that represent the next steps in computation. *Backward* analyses are useful for determining what happens after an event of interest. Live variables is a backward analysis that determines whether the value held in a variable may be subsequently

backward analysis

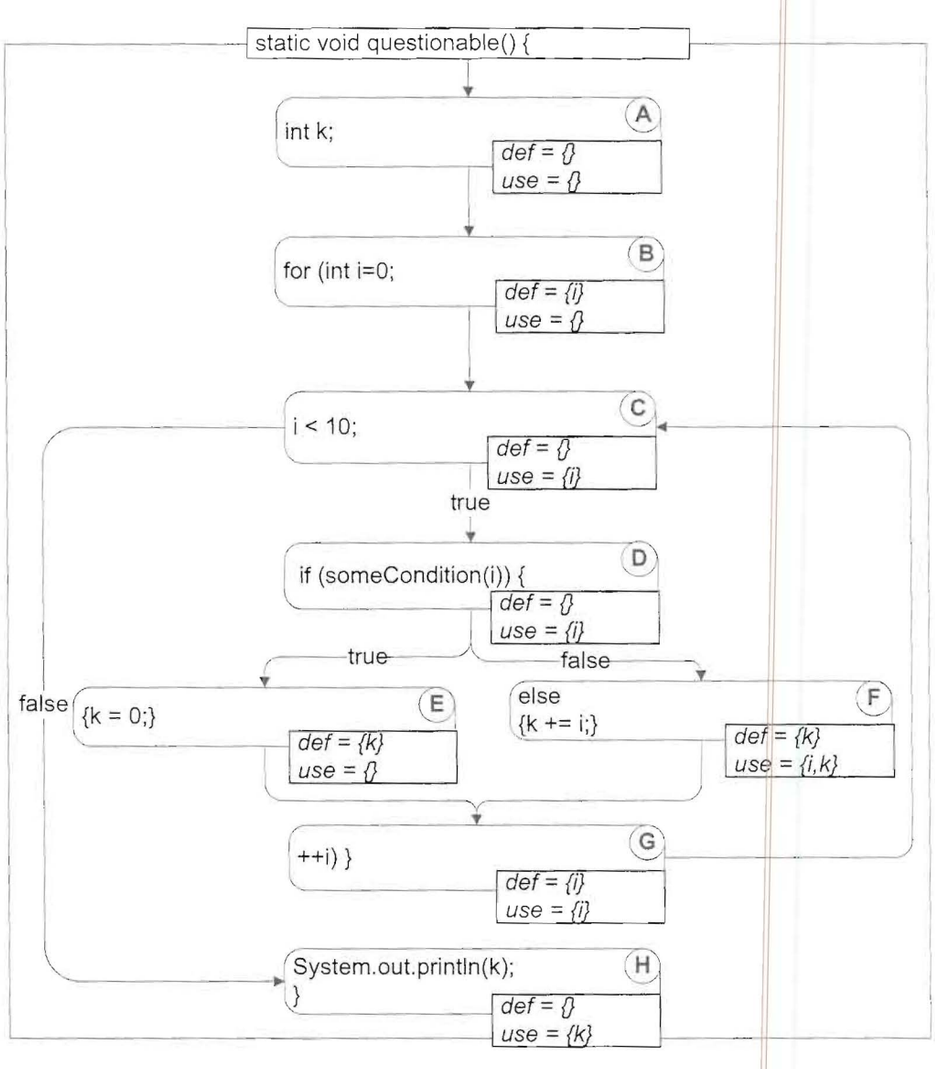

Figure 6.9: Control flow graph of the source code in Figure 6.8, annotated with variable definitions and uses.