necessarily imply that the stated property is indeed satisfied in all cases. Hence the need for testing. Despite the positive verdict by the model checker, testing is necessary to ascertain at least for a given set of situations that the application indeed satisfies the property.

While both model-checking and model-based testing use models, model checking uses finite-state models augmented with local properties that must hold at individual states. The local properties are known as atomic propositions and the augmented models as Kripke structures.

In summary, model-checking is to be viewed as a powerful and complementary technique to model-based testing. Neither can guarantee whether an application satisfies a property under all input conditions. However, both point to useful information that helps a tester discover subtle errors.

1.14 CONTROL-FLOW GRAPH

A CFG captures the flow of control within a program. Such a graph assists testers in the analysis of a program to understand its behavior in terms of the flow of control. A CFG can be constructed manually without much difficulty for relatively small programs, say containing less than about 50 statements. However, as the size of the program grows, so does the difficulty of constructing its CFG and hence arises the need for tools.

A CFG is also known by the names *flow graph* or *program graph*. However, it is not to be confused with the program-dependence graph (PDG) introduced in Section 1.16. In the remainder of this section we explain what a CFG is and how to construct one for a given program.

1.14.1 Basic Block

Let P denote a program written in a procedural programming language, be it high level as C or Java or a low level such as the 80×86 assembly. A *basic block*, or simply a *block*, in P is a sequence of consecutive statements with a single entry and a single exit point. Thus, a block has unique entry and exit points. These points are the first and the last statements within a basic block. Control always enters a basic block at its entry point and exits from its exit point. There is no possibility of exit or a halt at any point inside the basic block except at its exit point. The entry and exit points of a basic block coincide when the block contains only one statement.

Example 1.23: The following program takes two integers `x` and `y` and outputs x^y. There are a total of 17 lines in this program including the `begin` and `end`. The execution of this program begins at line 1 and moves through lines 2, 3, and 4 to line 5 containing an `if` statement. Considering that there is a decision at line 5, control could go to one of two possible destinations at lines 6 and 8. Thus, the sequence of statements starting at line 1 and ending at line 5 constitutes a basic block. Its only entry point is at line 1 and the only exit point is at line 5.

Program P1.2

```
 1 begin
 2   int x, y, power;
 3   float z;
 4   input (x, y);
 5   if (y<0)
 6     power=-y;
 7   else
 8     power=y;
 9   z=1;
10   while (power!=0){
11     z=z*x;
12   power=power-1;
13   }
14   if (y<0)
15     z=1/z;
16   output(z);
17 end
```

A list of all basic blocks in Program P1.2 is given below.

Block	Lines	Entry Point	Exit Point
1	2, 3, 4, 5	1	5
2	6	6	6
3	8	8	8
4	9	9	9
5	10	10	10
6	11, 12	11	12
7	14	14	14
8	15	15	15
9	16	16	16

Program P1.2 contains a total of nine basic blocks numbered sequentially from 1 to 9. Note how the `while` at line 10 forms a block of its own. Also note that we have ignored lines 7 and 13 from the listing because these are syntactic markers, and so are `begin` and `end` that are also ignored.

Note that some tools for program analyses place a procedure call statement in a separate basic block. If we were to do that, then we will place the `input` and `output` statements in Program P1.2 in two separate basic blocks. Consider the following sequence of statements extracted from Program P1.2.

```
1 begin
2   int x, y, power;
3   float z;
4   input (x, y);
5   if(y<0)
```

In the previous example, lines 1 through 5 constitute one basic block. The above sequence contains a call to the `input` function. If function calls are treated differently, then the above sequence of statements contains three basic blocks, one composed of lines 1 through 3, the second composed of line 4, and the third composed of line 5.

Function calls are often treated as blocks of their own because they cause the control to be transferred away from the currently executing function and hence raise the possibility of abnormal termination of the program. In the context of flow graphs, unless stated otherwise, we treat calls to functions like any other sequential statement that is executed without the possibility of termination.

1.14.2 Flow Graph: Definition and Pictorial Representation

A flow graph G is defined as a finite set N of nodes and a finite set E of directed edges. An edge (i, j) in E, with an arrow directed from i to j, connects nodes n_i and n_j in N. We often write $G = (N, E)$ to denote a flow graph G with nodes in N and edges in E. `Start` and `End` are two special nodes in N and are known as distinguished nodes. Every other node in G is reachable from `Start`. Also, every node in N has a path terminating at `End`. Node `Start` that has no incoming edge, and `End` that has no outgoing edge.

In a flow graph of program P, we often use a basic block as a node and edges indicate the flow of control across basic blocks. We label the blocks and the nodes such that block b_i corresponds to node n_i. An edge (i, j) connecting basic blocks b_i and b_j implies that control can go from block b_i to block b_j. Sometimes we will use a flow graph with one node corresponding to each statement in P.

A pictorial representation of a flow graph is often used in the analysis of the control behavior of a program. Each node is represented by a symbol, usually an oval or a rectangular box. These boxes are labeled by their corresponding block numbers. The boxes are connected by lines representing edges. Arrows are used to indicate the direction of flow. A block that ends in a decision has two edges going out of it. These edges are labeled `true` and `false` to indicate the path taken when the condition evaluates to `true` and `false`, respectively.

Example 1.24: The flow graph for Program P1.2 is defined as follows:

$N = \{$`Start, 1, 2, 3, 4, 5, 6, 7, 8, 9, End`$\}$

$E = \{$`(Start, 1), (1, 2), (1, 3), (2, 4), (3, 4), (4, 5), (5, 6), (6, 5), (5, 7), (7, 8), (7, 9), (9, End)`$\}$

Figure 1.16(a) depicts this flow graph. Block numbers are placed right next to or above the corresponding box. As shown in Figure 1.16(b), the contents of a block may be omitted, and nodes represented by circles, if we are interested only in the flow of control across program blocks and not their contents.

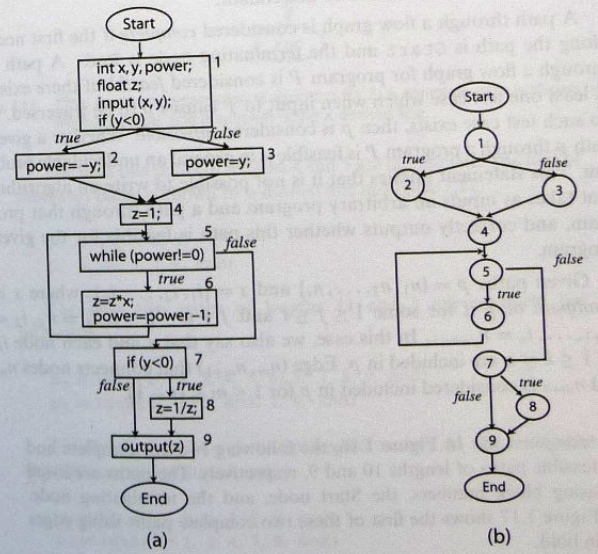

Fig. 1.16 Flow graph representations of Program P1.2. (a) Statements in each block are shown. (b) Statements within a block are omitted.

1.14.3 PATH

Consider a flow graph $G = (N, E)$. A sequence of k edges, $k \geq 0$, $(e_1, e_2, \ldots, e_k)$, denotes a path of length k through the flow graph if the following sequence condition holds: Given that n_p, n_q, n_r, and n_s are nodes belonging to N, and $0 < i < k$, if $e_i = (n_p, n_q)$ and $e_{i+1} = (n_r, n_s)$ then $n_q = n_r$.

Thus, for example, the sequence ((1, 3), (3, 4), (4, 5)) is a path in the flow graph shown in Figure 1.16. However, ((1, 3), (3, 5), 6, 8)) is not a valid path through this flow graph. For brevity, we indicate a path as a sequence of blocks. For example, in Figure 1.16, the edge sequence ((1, 3), (3, 4), (4, 5)) is the same as the block sequence (1, 3, 4, 5).

For nodes $n, m \in N$, m is said to be a *descendant of* n if there is a path from n to m; in this case n is an *ancestor* of m and m its descendant. If, in addition, $n \neq m$, then n is a *proper anscestor* of m, and m a *proper descendant of* n. If there is an edge $(n, m) \in E$, then m is a *successor* of n and n the *predecessor* of m. The set of all successor and predecessor nodes of n will be denoted by *succ(n)* and *pred(n)*, respectively. `Start` has no ancestor and `End` has no descendant.

A path through a flow graph is considered *complete* if the first node along the path is `Start` and the terminating node is `End`. A path p through a flow graph for program P is considered *feasible* if there exists at least one test case which when input to P causes p to be traversed. If no such test case exists, then p is considered *infeasible*. Whether a given path p through a program P is feasible is in general an undecidable problem. This statement implies that it is not possible to write an algorithm that takes as inputs an arbitrary program and a path through that program, and correctly outputs whether this path is feasible for the given program.

Given paths $p = \{n_1, n_2, \ldots, n_t\}$ and $s = \{i_1, i_2, \ldots, i_u\}$, where s is a *subpath* of p if for some $1 \leq j \leq t$ and $j + u - 1 \leq t$, $i_1 = n_j, i_2 = n_{j+1}, \ldots, i_u = n_{j+u-1}$. In this case, we also say that s and each node i_k for $1 \leq k \leq u$ are included in p. Edge (n_m, n_{m+1}) that connects nodes n_m and n_{m+1} is considered included in p for $1 \leq m \leq (t-1)$.

Example 1.25: In Figure 1.16, the following two are complete and feasible paths of lengths 10 and 9, respectively. The paths are listed using block numbers, the Start node, and the terminating node. Figure 1.17 shows the first of these two complete paths using edges in bold.

```
p1 = (Start, 1, 2, 4, 5, 6, 5, 7, 8, 9, End)
p2 = (Start, 1, 3, 4, 5, 6, 5, 7, 9, End)
```

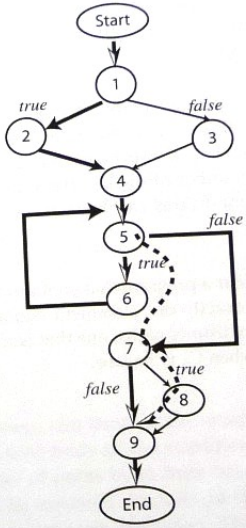

Fig. 1.17 Flow graph representation of Program P1.2. A complete path is shown using bold edges and a subpath using a dashed line.

The next two paths of lengths 4 and 5 are incomplete. The first of these two paths is shown by a dashed line in Figure 1.17.

```
p3 = (5, 7, 8, 9)
p4 = (6, 5, 7, 9, End)
```

The next two paths of lengths 11 and 8 are complete but infeasible.

```
p5 = (Start, 1, 3, 4, 5, 6, 5, 7, 8, 9, End)
p6 = (Start, 1, 2, 4, 5, 7, 9, End)
```

Finally, the next two paths are invalid as they do not satisfy the sequence condition stated earlier.

```
p7 = (Start, 1, 2, 4, 8, 9, End)
p8 = (Start, 1, 2, 4, 7, 9, End)
```

Nodes 2 and 3 in Figure 1.17 are successors of node 1, nodes 6 and 7 are successors of node 5, and nodes 8 and 9 are successors of node 7. Nodes 6, 7, 8, 9, and `End` are descendants of node 5. We also

have $succ(5) = \{5, 6, 7, 8, 9, \texttt{End}\}$ and $pred(5) = \{\texttt{Start}, 1, 2, 3, 4, 5, 6\}$. Note that in the presence of loops, a node can be its own ancestor or descendant.

There can be many distinct paths through a program. A program with no condition contains exactly one path that begins at node `Start` and terminates at node `End`. However, each additional condition in the program increases the number of distinct paths by at least one. Depending on their location, conditions can have an exponential effect on the number of paths.

Example 1.26: Consider a program that contains the following statement sequence with exactly one statement containing a condition. This program has two distinct paths, one that is traversed when C_1 is `true` and the other when C_1 is `false`.

```
begin
  S1;
  S2;
  :
  if (C1) {...}
  :
  Sn;
end
```

We modify the program given above by adding another `if`. The modified program, shown below, has exactly four paths that correspond to four distinct combinations of conditions C_1 and C_2.

```
begin
  S1;
  S2;
  :
  if (C1) {...}
  :
  if (C2) {...}
  Sn;
end
```

Note the exponential effect of adding an `if` on the number of paths. However, if a new condition is added within the scope of an `if` statement then the number of distinct paths increases only by one as is the case in the following program which has only three distinct paths.

```
begin
  S1;
  S2;
  :
  if (C1) {
  :
  if (C2) {...}
  :
  }
  :
  Sn;
end
```

The presence of loops can enormously increase the number of paths. Each traversal of the loop body adds a condition to the program, thereby increasing the number of paths by at least one. Sometimes, the number of times a loop is to be executed depends on the input data and cannot be determined prior to program execution. This becomes another cause of difficulty in determining the number of paths in a program. Of course, one can compute an upper limit on the number of paths based on some assumption on the input data.

Example 1.27: Program P1.3 inputs a sequence of integers and computes their product. A Boolean variable `done` controls the number of integers to be multiplied. A flow graph for this program appears in Figure 1.18.

Program P1.3

```
1 begin
2   int num, product, power;
3   bool done;
4   product=1;
5   input(done);
6   while (!done){
7     input(num);
8     product=product * num;
9     input(done);
10  }
11  output(product);
12 end
```

As shown in Figure 1.18, Program P1.3 contains four basic blocks and one condition that guards the body of `while`. (`Start`, 1, 2,

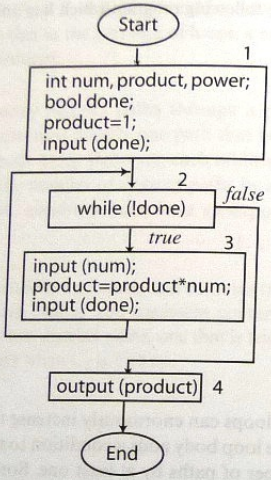

Fig. 1.18 Flow graph of Program P1.3. Numbers 1 through 4 indicate the four basic blocks in Program P1.3.

4, End) is the path traversed when done is true the first time the loop condition is checked. If there is only one value of num to be processed, then the path followed is (Start, 1, 2, 3, 2, 4, End). When there are two input integers to be multiplied then the path traversed is (Start, 1, 2, 3, 2, 3, 2, 4, End).

Notice that the length of the path traversed increases with the number of times the loop body is traversed. Also, the number of distinct paths in this program is the same as the number of different lengths of input sequences that are to be multiplied. Thus, when the input sequence is empty, that is of length 0, the length of the path traversed is 4. For an input sequence of length 1, it is 6, for 2 it is 8, and so on.

1.15 DOMINATORS AND POSTDOMINATORS

Let G = (N, E) be a CFG for program P. Recall that G has two special nodes labeled Start and End. We define the dominator and postdominator as two relations on the set of nodes N. These relations find various applications, especially during the construction of tools for test adequacy assessment (Chapter 6) and regression testing (Chapter 5).

For nodes n and m in N, we say that n dominates m if n lies on every path from Start to m. We write $dom(n, m)$ when n dominates m. In an analogous manner, we say that node n postdominates node m if n lies on every path from m to the End node. We write $pdom(n, m)$ when n postdominates m. When $n \neq m$, we refer to these as strict dominator and strict postdominator relations. $dom(n)$ and $pdom(n)$ denote the sets of dominators and postdominators of node n, respectively.

For $n, m \in N$, n is the immediate dominator of m when n is the last dominator of m along a path from the Start to m. We write $idom(n, m)$ when n is the immediate dominator of m. Each node, except for Start, has a unique immediate dominator. Immediate dominators are useful in that we can use them to build a dominator tree. A dominator tree derived from G succinctly shows the dominator relation.

For $n, m \in N$, n is an immediate postdominator of m if n is the first postdominator along a path from m to End. Each node, except for End, has a unique immediate postdominator. We write $ipdom(n, m)$ when n is the immediate postdominator of m. Immediate postdominators allow us to construct a postdominator tree that exhibits the postdominator relation among the nodes in G.

Example 1.28: Consider the flow graph in Figure 1.18. This flow graph contains six nodes including Start and End. Its dominators and postdominators are shown in Figure 1.19(a) and (b), respectively. In the dominator tree, for example, a directed edge connects

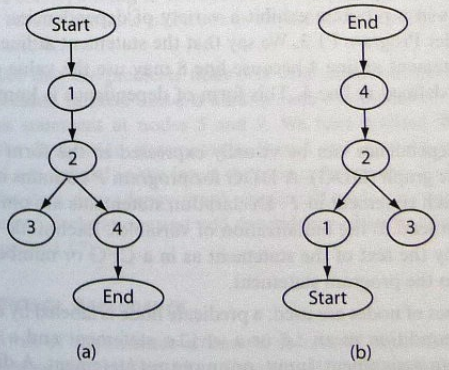

Fig. 1.19 (a) Dominator and (b) postdominator trees derived from the flow graph in Figure 1.18.

an immediate dominator to the node it dominates. Thus, among other relations, we have *idom*(1, 2) and *idom*(4, `end`). Similarly, from the postdominator tree, we obtain *ipdom*(4, 2) and *ipdom* (`End`, 4).

Given a dominator and a postdominator tree, it is easy to derive the set of dominators and postdominators for any node. For example, the set of dominators for node 4, denoted as *dom*(4), is {2, 1, `Start`}. *dom*(4) is derived by first obtaining the immediate dominator of 4 which is 2, followed by the immediate dominator of 2 which is 1, and finally the immediate dominator of 1 which is `Start`. Similarly, we can derive the set of postdominators for node 2 as {4, `End`}.

1.16 PROGRAM-DEPENDENCE GRAPH

A PDG for program P exhibits different kinds of dependencies among statements in P. For the purpose of testing, we consider data dependence and control dependence. These two dependencies are defined with respect to data and predicates in a program. Next, we explain data and control dependences, how they are derived from a program and their representation in the form of a PDG. We first show how to construct a PDG for programs with no procedures and then show how to handle programs with procedures.

1.16.1 Data Dependence

Statements in a program exhibit a variety of dependencies. For example, consider Program P1.3. We say that the statement at line 8 depends on the statement at line 4 because line 8 may use the value of variable `product` defined at line 4. This form of dependence is known as *data dependence.*

Data dependence can be visually expressed in the form of a data-dependence graph (DDG). A DDG for program P contains one unique node for each statement in P. Declaration statements are omitted when they do not lead to the initialization of variables. Each node in a DDG is labeled by the text of the statement as in a CFG or numbered corresponding to the program statement.

Two types of nodes are used: a predicate node is labeled by a predicate such as a condition in an `if` or a `while` statement and a data node labeled by an assignment, input, or an output statement. A directed arc from node n_2 to n_1 indicates that node n_2 is data dependent on node

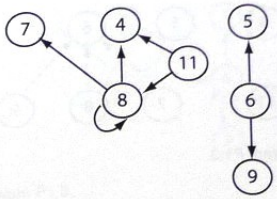

Fig. 1.20 DDG for Program P1.3. Node numbers correspond to line numbers. Declarations have been omitted.

n_1. This kind of data dependence is also known as flow dependence. A definition of data dependency follows:

> **DATA DEPENDENCE**
>
> *Let D be a DDG with nodes n_1 and n_2. Node n_2 is data dependent on n_1 if (a) variable v is defined at n_1 and used at n_2 and (b) there exists a path of nonzero length from n_1 to n_2 not containing any node that redefines v.*

Example 1.29: Consider Program P1.3 and its DDG in Figure 1.20. The graph shows seven nodes corresponding to the program statements. Data dependence is exhibited using directed edges. For example, node 8 is data dependent on nodes 4, 7, and itself because variable `product` is used at node 8 and defined at nodes 4 and 8, and variable `num` is used at node 8 and defined at node 7. Similarly, node 11, corresponding to the output statement, depends on nodes 4 and 8 because variable `product` is used at node 11 and defined at nodes 4 and 8.

Notice that the predicate node 6 is data dependent on nodes 5 and 9 because variable `done` is used at node 6 and defined through an input statement at nodes 5 and 9. We have omitted from the graph the declaration statements at lines 2 and 3 as the variables declared are defined in the input statements before use. To be complete, the data dependency graph could add nodes corresponding to these two declarations and add dependency edges to these nodes (see Exercise 1.17).

1.16.2 Control Dependence

Another form of dependence is known as *control dependence.* For example, the statement at line 12 in Program P1.2 depends on the predicate at line 10. This is because control may or may not arrive at line

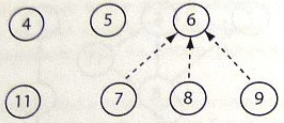

Fig. 1.21 CDG for Program P1.3.

12 depending on the outcome of the predicate at line 10. Note that the statement at line 9 does not depend on the predicate at line 5 because control will arrive at line 9 regardless of the outcome of this predicate.

As with data dependence, control dependence can be visually represented as a control-dependence graph (CDG). Each program statement corresponds to a unique node in the CDG. There is a directed edge from node n_2 to n_1 when n_2 is control dependent on n_1.

CONTROL DEPENDENCE

Let C be a CDG with nodes n_1 and n_2. n_1 being a predicate node. Node n_2 is control dependent on n_1 if there is at least one path from n_1 to program exit that includes n_2 and at least one path from n_1 to program exit that excludes n_2.

Example 1.30: Figure 1.21 shows the CDG for program P1.3. Control dependence edges are shown as dotted lines.

As shown, nodes 7, 8, and 9 are control dependent on node 6 because there exists a path from node 6 to each of the dependent nodes as well as a path that excludes these nodes. Notice that none of the remaining nodes is control dependent on node 6, the only predicate node in this example. Node 11 is not control dependent on node 6 because any path that goes from node 6 to program exit includes node 11.

Now that we know what data and control dependence is, we can show the PDG as a combination of the two dependencies. Each program statement contains one node in the PDG. Nodes are joined with directed arcs showing data and control dependence. A PDG can be considered as a combination of two subgraphs: a data dependence subgraph and a control-dependence subgraph.

Example 1.31: Figure 1.22 shows the PDG for Program P1.3. It is obtained by combining the graphs shown in Figures 1.20 and 1.21 (see Exercise 1.18).

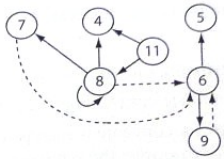

Fig. 1.22 PDG for Program P1.3.

1.17 STRINGS, LANGUAGES, AND REGULAR EXPRESSIONS

Strings play an important role in testing. As we shall see in Section 3.2, Chapter 3, strings serve as inputs to a FSM and hence to its implementation as a program. Thus a string serves as a test input. A collection of strings also forms a language. For example, a set of all strings consisting of zeros and ones is the language of binary numbers. In this section we provide a brief introduction to strings and languages.

A collection of symbols is known as an *alphabet*. We will use uppercase letters such as X and Y to denote alphabets. Though alphabets can be infinite, we are concerned only with finite alphabets. For example, $X = \{0, 1\}$ is an alphabet consisting of two symbols 0 and 1. Another alphabet is $Y = \{\text{dog, cat, horse, lion}\}$ that consists of four symbols *dog*, *cat*, *horse*, and *lion*.

A string over an alphabet X is any sequence of zero or more symbols that belong to X. For example, 0110 is a string over the alphabet $\{0, 1\}$. Also, `dog cat dog dog lion` is a string over the alphabet {*dog, cat, horse, lion*}. We will use lowercase letters such as p, q, r to denote strings. The length of a string is the number of symbols in that string. Given a string s, we denote its length by $|s|$. Thus, $|1011| = 4$ and $|$`dog cat dog`$| = 3$. A string of length 0, also known as an *empty string*, is denoted by ϵ.

Let s_1 and s_2 be two strings over alphabet X. We write $s_1 \cdot s_2$ to denote the *concatenation* of strings s_1 and s_2. For example, given the alphabet $X = \{0, 1\}$, and two strings 011 and 101 over X, we obtain $011 \cdot 101 = 011101$. It is easy to see that $|s_1 \cdot s_2| = |s_1| + |s_2|$. Also, for any string s, we have $s \cdot \epsilon = s$ and $\epsilon \cdot s = s$.

A set L of strings over an alphabet X is known as a *language*. A language can be finite or infinite. Given languages L_1 and L_2, we denote their catenation as $L_1 \cdot L_2$ that denotes the set L defined as:

$$L = L_1 \cdot L_2 = \{x \cdot y \mid x \in L_1, y \in L_2\}$$

Even if a unique target is determined for each procedure invocation, inlining must be applied judiciously. In general, it is not possible to inline recursive procedures directly, and even without recursion, inlining can expand the code size exponentially.

12.1 Basic Concepts

In this section, we introduce call graphs — graphs that tell us which procedures can call which. We also introduce the idea of "context sensitivity," where data-flow analyses are required to take cognizance of what the sequence of procedure calls has been. That is, context-sensitive analysis includes (a synopsis of) the current sequence of activation records on the stack, along with the current point in the program, when distinguishing among different "places" in the program.

12.1.1 Call Graphs

A *call graph* for a program is a set of nodes and edges such that

1. There is one node for each procedure in the program.
2. There is one node for each *call site*, that is, a place in the program where a procedure is invoked.
3. If call site c may call procedure p, then there is an edge from the node for c to the node for p.

Many programs written in languages like C and Fortran make procedure calls directly, so the call target of each invocation can be determined statically. In that case, each call site has an edge to exactly one procedure in the call graph. However, if the program includes the use of a procedure parameter or function pointer, the target generally is not known until the program is run and, in fact, may vary from one invocation to another. Then, a call site can link to many or all procedures in the call graph.

Indirect calls are the norm for object-oriented programming languages. In particular, when there is overriding of methods in subclasses, a use of method m may refer to any of a number of different methods, depending on the subclass of the receiver object to which it was applied. The use of such *virtual* method invocations means that we need to know the type of the receiver before we can determine which method is invoked.

Example 12.1: Figure 12.1 shows a C program that declares `pf` to be a global pointer to a function whose type is "integer to integer." There are two functions of this type, `fun1` and `fun2`, and a main function that is not of the type that `pf` points to. The figure shows three call sites, denoted `c1`, `c2`, and `c3`; the labels are not part of the program.


```
        int (*pf)(int);

        int fun1(int x) {
            if (x < 10)
c1:             return (*pf)(x+1);
            else
                return x;
        }

        int fun2(int y) {
            pf = &fun1;
c2:         return (*pf)(y);
        }

        void main() {
            pf = &fun2;
c3:         (*pf)(5);
        }
```

Figure 12.1: A program with a function pointer

The simplest analysis of what `pf` could point to would simply observe the types of functions. Functions `fun1` and `fun2` are of the same type as what `pf` points to, while `main` is not. Thus, a conservative call graph is shown in Fig. 12.2(a). A more careful analysis of the program would observe that `pf` is made to point to `fun2` in `main` and is made to point to `fun1` in `fun2`. But there are no other assignments to any pointer, so, in particular, there is no way for `pf` to point to `main`. This reasoning yields the same call graph as Fig. 12.2(a).

An even more precise analysis would say that at `c3`, it is only possible for `pf` to point to `fun2`, because that call is preceded immediately by that assignment to `pf`. Similarly, at `c2` it is only possible for `pf` to point to `fun1`. As a result, the initial call to `fun1` can come only from `fun2`, and `fun1` does not change `pf`, so whenever we are within `fun1`, `pf` points to `fun1`. In particular, at `c1`, we can be sure `pf` points to `fun1`. Thus, Fig. 12.2(b) is a more precise, correct call graph. □

In general, the presence of references or pointers to functions or methods requires us to get a static approximation of the potential values of all procedure parameters, function pointers, and receiver object types. To make an accurate approximation, interprocedural analysis is necessary. The analysis is iterative, starting with the statically observable targets. As more targets are discovered, the analysis incorporates the new edges into the call graph and repeats discovering more targets until convergence is reached.

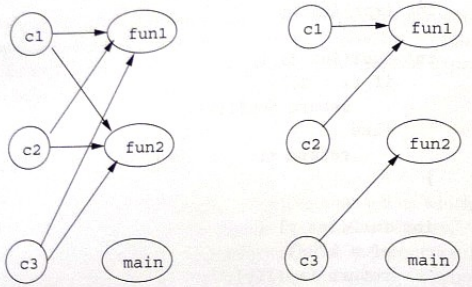

Figure 12.2: Call graphs derived from Fig. 12.1

12.1.2 Context Sensitivity

Interprocedural analysis is challenging because the behavior of each procedure is dependent upon the context in which it is called. Example 12.2 uses the problem of interprocedural constant propagation on a small program to illustrate the significance of contexts.

Example 12.2: Consider the program fragment in Fig. 12.3. Function f is invoked at three call sites: `c1`, `c2` and `c3`. Constant 0 is passed in as the actual parameter at `c1`, and constant 243 is passed in at `c2` and `c3` in each iteration; the constants 1 and 244 are returned, respectively. Thus, function f is invoked with a constant in each of the contexts, but the value of the constant is context-dependent.

As we shall see, it is not possible to tell that `t1`, `t2`, and `t3` each are assigned constant values (and thus so is $X[i]$), unless we recognize that when called in context `c1`, f returns 1, and when called in the other two contexts, f returns 244. A naive analysis would conclude that f can return either 1 or 244 from any call. □

One simplistic but extremely inaccurate approach to interprocedural analysis, known as *context-insensitive analysis*, is to treat each call and return statement as "goto" operations. We create a *super* control-flow graph where, besides the normal intraprocedural control flow edges, additional edges are created connecting

1. Each call site to the beginning of the procedure it calls, and
2. The return statements back to the call sites.[1]

[1]The return is actually to the instruction following the call site.


```
        for (i = 0; i < n; i++) {
c1:         t1 = f(0);
c2:         t2 = f(243);
c3:         t3 = f(243);
            X[i] = t1+t2+t3;
        }

        int f (int v) {
            return (v+1);
        }
```

Figure 12.3: A program fragment illustrating the need for context-sensitive analysis

Assignment statements are added to assign each actual parameter to its corresponding formal parameter and to assign the returned value to the variable receiving the result. We can then apply a standard analysis intended to be used within a procedure to the super control-flow graph to find context-insensitive interprocedural results. While simple, this model abstracts out the important relationship between input and output values in procedure invocations, causing the analysis to be imprecise.

Example 12.3: The super control-flow graph for the program in Fig. 12.3 is shown in Figure 12.4. Block B_6 is the function f. Block B_3 contains the call site `c1`; it sets the formal parameter v to 0 and then jumps to the beginning of f, at B_6. Similarly, B_4 and B_5 represent the call sites `c2` and `c3`, respectively. In B_4, which is reached from the end of f (block B_6), we take the return value from f and assign it to `t1`. We then set formal parameter v to 243 and call f again, by jumping to B_6. Note that there is no edge from B_3 to B_4. Control must flow through f on the way from B_3 to B_4.

B_5 is similar to B_4. It receives the return from f, assigns the return value to `t2`, and initiates the third call to f. Block B_7 represents the return from the third call and the assignment to $X[i]$.

If we treat Fig. 12.4 as if it were the flow graph of a single procedure, then we would conclude that coming into B_6, v can have the value 0 or 243. Thus, the most we can conclude about `retval` is that it is assigned 1 or 244, but no other value. Similarly, we can only conclude about `t1`, `t2`, and `t3` that they can each be either 1 or 244. Thus, $X[i]$ appears to be either 3, 246, 489, or 732. In contrast, a context-sensitive analysis would separate the results for each of the calling contexts and produces the intuitive answer described in Example 12.2: `t1` is always 1, `t2` and `t3` are always 244, and $X[i]$ is 489. □