


# control-flow graph

---

 [sciencedirect.com/topics/computer-science/control-flow-graph](https://www.sciencedirect.com/topics/computer-science/control-flow-graph)

## Deducing Errors

---

Andreas Zeller, in *Why Programs Fail (Second Edition)*, 2009

### 7.3 TRACKING DEPENDENCES

---

The control flow graph is the basis for all deduction about programs, as it shows how information propagates along the sequence of statements. Let's go a little more into detail here. Exactly how do individual statements affect the information flow? And how are statements affected by the information flow?

#### 7.3.1 Effects of Statements

---

To contribute to the computation, every statement of the program must (at least potentially) affect the information flow in some way. We distinguish the following two types of effects.

##### Write

A statement can *change the program state* (i.e., assign a value to a variable). For example, the statement `v1 = 1` writes a value to the variable `v1`.

The “program state” considered here is a very general term. For instance, printing some text on an output device changes the state of the device. Sending a message across the network changes the state of attached devices. To some extent, the “program state” thus becomes the state of the world. Therefore, it is useful to limit the considered state—for instance, to the hardware boundaries.

##### Control

A statement may *change the program counter*—that is, determine which statement is to be executed next. In Figure 7.1, the while statement determines whether the next statement is either 5 or 9. Obviously, we are only talking about *conditional* changes to the program counter here—that is, statements that have at least two possible successors in the control flow graph, dependent on the program state.

In principle, one may consider the program counter as part of the program state. In practice, though, locations in the program state, and locations in the program code, are treated conceptually as separate dimensions of space and time. Figure 1.1 uses this distinction to represent the intuition about what is happening in a program run.

#### 7.3.2 Affected Statements

Affecting the information flow by writing state or controlling execution represents the *active* side of how a statement affects the information flow. However, statements are also *passively* affected by other statements. For example:

■

*Read.* A statement can *read the program state* (i.e., read a value from a variable). For example, the statement  $v2 = v1 + 1$  reads a value from the variable  $v1$ . Consequently, the effect of the statement is affected by the state of  $v1$ .

Just as in the writing state, the “program state” considered here is a very general term. For instance, reading some text from an input device reads the state of the device. Reading a message across the network reads the state of attached devices.

■

*Execution.* To have any effect, a statement must be *executed*. Consequently, if the execution of a statement  $B$  is potentially controlled by another statement  $A$ , then  $B$  is affected by  $A$ .

For each statement  $S$  of a program, we can determine what part of the state is being read or written by  $S$  (as deduced from the actual program code), and which other statements are controlled by  $S$  (as deduced from the control flow graph). As an example, consider Table 7.1, which lists the actions of the statements in the `fib()` program.

Table 7.1. Effects of the `fib()` Statements

	<b>Statement</b>	<b>Reads</b>	<b>Writes</b>	<b>Controls</b>
0	<code>fib(n)</code>		$n$	1–10
1	<code>int f</code>		$f$	
2	<code>f0 = 1</code>		$f0$	
3	<code>f1 = 1</code>		$f1$	
4	<code>while (n &gt; 1)</code>	$n$		5–8
5	<code>n = n - 1</code>	$n$	$n$	
6	<code>f = f0 + f1</code>	$f0, f1$	$f$	
7	<code>f0 = f1</code>	$f1$	$f0$	
8	<code>f1 = f</code>	$f$	$f1$	
9	<code>return f</code>	$f$	return value	

Note: Each statement reads or writes a variable, or controls whether other statements are executed.

### 7.3.3 Statement Dependences

---

Given the effects of statements, as well as the statements thereby affected, we can construct *dependences* between statements, showing how they influence each other. We distinguish the following two types of dependences.

#### Data dependence

A statement  $B$  is *data dependent* on a statement  $A$  if

■

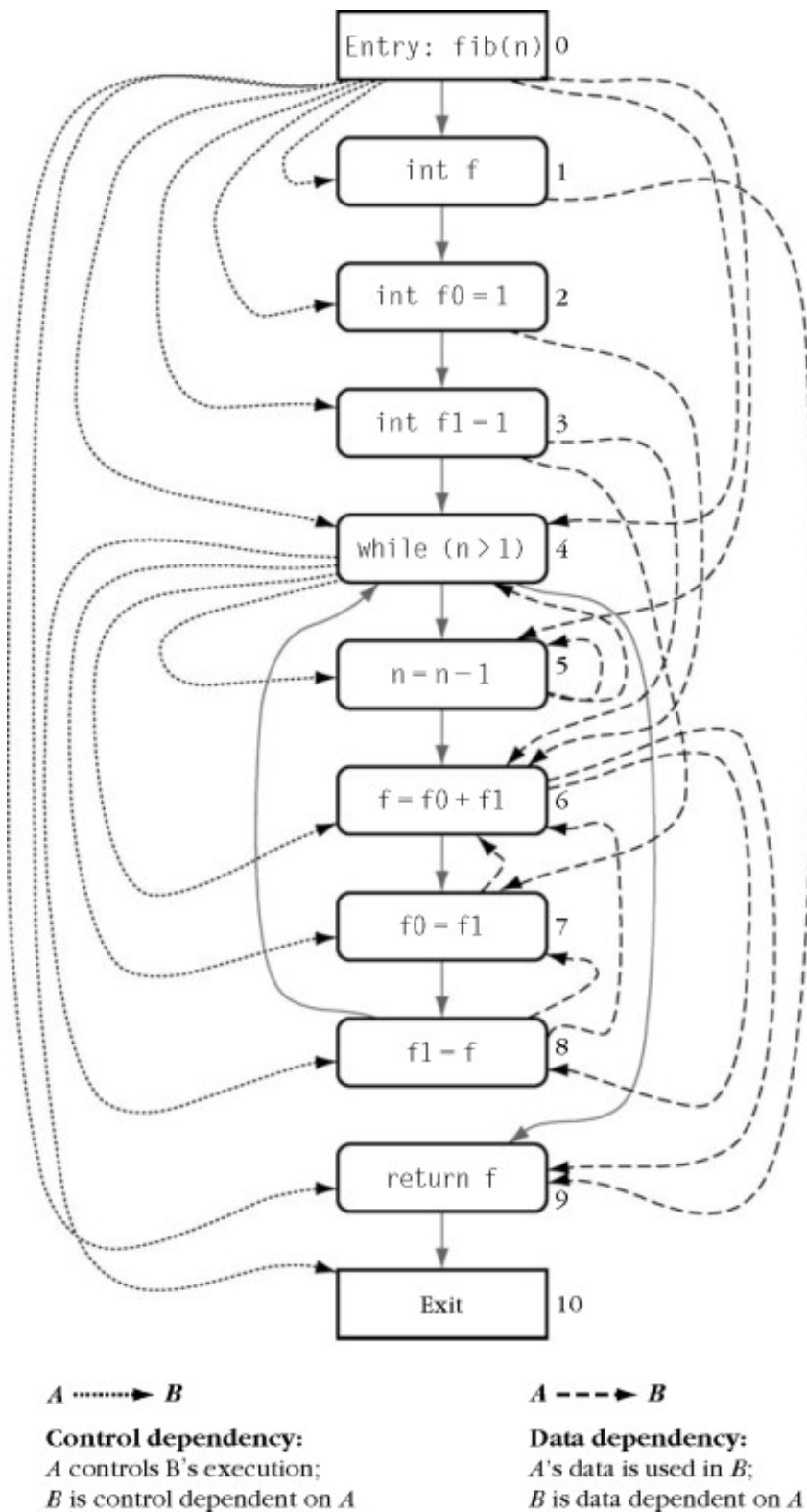
$A$  writes some variable  $V$  (or more generally, part of the program state) that is being read by  $B$ , and

■

there is at least one path in the control flow graph from  $A$  to  $B$  in which  $V$  is not being written by some other statement.

In other words, the outcome of  $A$  can influence the data read by  $B$ .

Figure 7.3 shows the data dependences in the `fib()` program. By following the dashed arrows on the right side, one can determine where the data being written by some statement are being read by another statement. For instance, the variable `f0` being written by statement 2, `f0 = 1`, is read in statement 6, `f = f0 + f1`.



[Sign in to download full-size image](#)  
 FIGURE 7.3. fib() dependence graph.

## Control dependence

A statement  $B$  is *control dependent* on a statement  $A$  if  $B$ 's execution is potentially controlled

by *A*. In other words, the outcome of *A* determines whether *B* is executed.

The dotted arrows on the left side of Figure 7.3 show the control dependences in `fib()`. Each statement of the body of the while loop is dependent on entering the loop (and thus dependent on the head of the while loop). All other statements are dependent on the entry of the function (as it determines whether the body is actually executed).

The control and data dependences of a program, as shown in Figure 7.3, form a graph—the *program-dependence graph*. This graph is the basis for a number of program-analysis techniques, as it reflects all influences within a program.

### 7.3.4 Following Dependences

---

Following the control and data dependences in the program-dependence graph, one can determine which statements influence which other statements—in terms of data or control, or both. In particular, one can answer two important questions:

#### 1.

*Where does this value go to?* Given a statement *S* writing a variable *V*, we can determine the impact of *S* by checking which other statements are dependent on *S*—and which other statements are dependent on these. Let's follow the dependences to see what happens when we call `fib()`. In Figure 7.3, the value of *n* is being used in the while head (statement 4) as well as in the while body (statement 5). Because the while head also controls the assignments to *f*, *f0*, and *f1* (statements 6–8), the value of *n* also determines the values of *f*, *f0*, and *f1*—and eventually, the returned value *f*. This is how `fib()` is supposed to work.

#### 2.

*Where does this value come from?* Given a statement *S* reading a variable *V*, we can determine the statements that possibly influenced *V* by following back the dependences of *S*. Let's now follow the dependences to see where the arbitrary value returned by `fib(1)` comes from. In Figure 7.3, consider the return value *f* in statement 9. The value of *f* can come from two sources. It can be computed in statement 6 from *f0* and *f1* (and, following their control dependences, eventually *n*). However, it also can come from statement 1, which is the declaration of *f*. In the C language, a local variable is not initialized and thus may hold an arbitrary value. It is exactly this value that is returned if the while body is not executed. In other words, this is the arbitrary value returned by `fib(1)`.

### 7.3.5 Leveraging Dependences

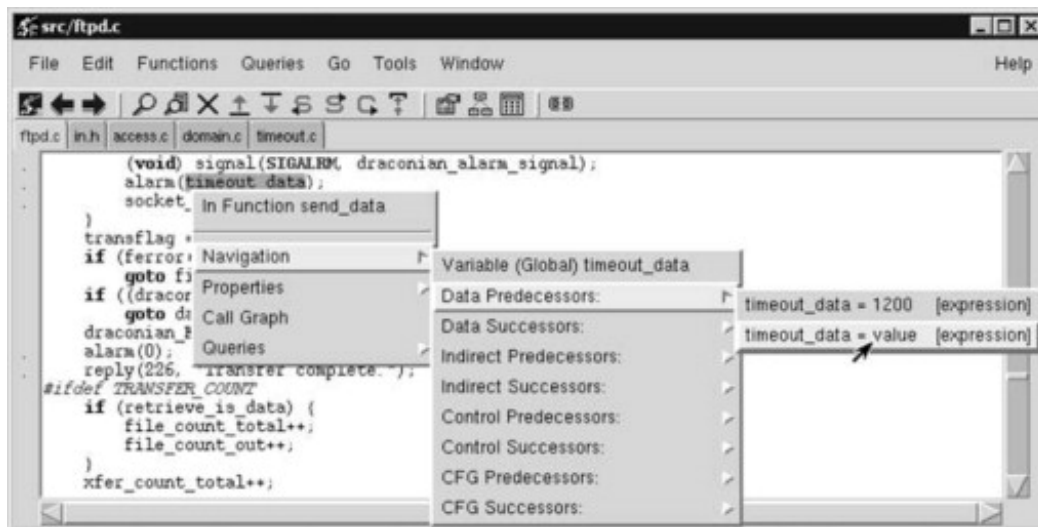
---

Following dependences through programs is a common technique for finding out the origins of values. But from where does one get the dependences?

Typically, programmers *implicitly* determine dependences while reading the code. Assessing the effect of each statement is part of understanding the program. Studies have shown that

programmers effectively follow dependences while debugging, either in a *forward* fashion to assess the impact of a statement or in a *backward* fashion to find out which other parts of the program might have influenced a statement. As Weiser (1982) puts it: "When debugging, programmers view programs in ways that need not conform to the programs' textual or modular structures." Thus, dependences become an important guide for *navigating through the code*.

Obtaining *explicit* dependences such that they can be leveraged in tools is also feasible and is part of several advanced program-analysis tools. Figure 7.4 shows a screenshot of the CODESURFER tool, one of the most advanced program-analysis tools available. Rather than visualizing dependences as in Figure 7.3, CODESURFER allows programmers to explore the dependences interactively by navigating to predecessors and successors according to data and control dependences.



[Sign in to download full-size image](#)

FIGURE 7.4. Following dependences in CODESURFER. For each variable, one can query the predecessors and successors along the dependence graph.

[View chapter](#) [Purchase book](#)

## Data-Flow Analysis

Keith D. Cooper, Linda Torczon, in *Engineering a Compiler (Second Edition)*, 2012

## Other Language Issues

In intraprocedural analysis, we assume that the control-flow graph has a single entry and a single exit; we add an artificial exit node if the procedure has multiple returns. In interprocedural analysis, language features can create the same kinds of problems.

For example, Java has both initializers and finalizers. The Java virtual machine invokes a class initializer after it loads and verifies the class; it invokes an object initializer after it allocates space for the object but before it returns the object's hashcode. Thread start methods, finalizers, and destructors also have the property that they execute without an explicit call in the source program.

The call-graph builder must pay attention to these procedures. Initializers may be connected to sites that create objects; finalizers might be connected to the call-graph's entry node. The specific connections will depend on the language definition and the analysis being performed. MayMod analysis, for example, might ignore them as irrelevant, while interprocedural constant propagation needs information from initialization and start methods.

[View chapter](#)[Purchase book](#)

## Input-Sensitive Profiling

---

A. Alourani, ... M. Grechanik, in *Advances in Computers*, 2016

### 3.3.1 Summary

---

Traditional profilers link performance metrics to nodes and paths in control flow graphs (or *call graphs*) by gathering performance measurements (eg, execution time) for specific input values to help developers improve the performance of software applications by identifying what methods consume more resources (eg, CPU and memory usage). However, these profiling techniques do not pinpoint why these methods are responsible for intensive resource usages and do not identify how the resource consumptions of the same method differ with the increasing size of the input (eg, the number of nodes in an input linked list or a tree or a bigger input array). That is, when executing an application with different sizes of inputs, the same method of this application often consumes different resources. A main problem with profiling techniques is that they do not explain how the cost that measures resource usage is affected individually by the size of the input, the algorithm (eg, recursions and loops), and the underlying implementation of algorithms (eg, traversing a data structure iteratively or recursively). Traditional profilers calculate resource usage by combining these factors and provide limited information by reporting the overall cost. It is increasingly important to find a way of identifying how individual factors, including the size of input, algorithm, and implementation, impact the cost to uncover the relationship of the execution cost to the program input.

Zaparanuks and Hauswirth propose an automated profiling methodology to help developers to detect algorithmic (eg, recursions and loops) inefficiencies by inferring a cost function of a program that relates the cost to the input size and to predict how the resource

usage would scale with increasing the size of the input. AlgoProf was developed to automatically identify algorithms (eg, recursions and loops) in a program and infer the time complexity of each algorithm for a specific algorithmic step (eg, the total number of loop iterations) during performance testing. An important feature of this profiling technique is the ability to pinpoint why methods are responsible for intensive resource usages (eg, the CPU and memory) and execution times. Aside from detecting the root causes of scalability problems in a program, this technique can address the problem of measuring the size of the input automatically.

The authors introduced an algorithmic profiler for computing the cost function of a program by identifying algorithms (eg, loops and recursions) and their inputs to measure their sizes (eg, the number of nodes in a linked list), costs (eg, the execution times of loop iterations), and generated performance plots that mapped input size to the cost, ie, they compute cost functions for individual algorithms (eg, loops and recursions). This technique allows developers to make an accurate estimate of the computational cost as a function of algorithms (eg, loops and recursions) based on multiple program runs. The profiling technique employs cost metrics based on a repetition data structure access, such as the execution times of loop iterations, as compared with the execution times of the whole method that is used by traditional profiling techniques. These traditional techniques provide a single cost value, such as hotness (eg, longer execution times), whereas the algorithmic profiler provides much deeper insight into a function that maps the cost to the size and type of the input. Thus, the algorithmic complexity can be inferred more accurately by using cost functions to detect algorithmic inefficiencies. The algorithmic profiler enables developers to understand how the resource usage measurements are affected by the size of the input, the algorithm, and the type of underlying implementation individually. Aside from identifying the root causes of scalability problems in a program, this technique pinpoints why methods are responsible for intensive resource usages (eg, the CPU and memory) and execution times.

The effectiveness of AlgoProf in detecting algorithmic (eg, recursions and loops) inefficiencies was evaluated with a number of programs that implement different algorithms (eg, recursions and loops). Every program uses one data structure type, eg, an array, a linked list, a tree, or a graph. AlgoProf was able to estimate the algorithmic complexities of all data structures in the programs accurately, along with inferring their cost functions to detect algorithmic inefficiencies. Furthermore, the ability of AlgoProf to identify the root causes of scalability problems was evaluated using a Java program that requires the assignment of a larger array when the size of array runs out of space. AlgoProf shows a plot that links a cost (eg, execution times) with the growing array. If the array is grown by a single element at a time, the cost becomes quadratic or worse, ie, exponential. If the array is grown by doubling the size and changing a single line of the source code, the cost can be reduced to a linear function. To sum up, the proposed profiling technique proves to have the ability to help developers detect algorithmic inefficiencies and pinpoint why methods



are responsible for intensive resource usage (eg, the CPU and memory) and execution time. Moreover, AlgoProf provided an effective performance profiling solution to estimate the time complexity and detect algorithmic inefficiencies of a method in a program.

[View chapter](#)[Purchase book](#)

## Register Allocation

---

*Keith D. Cooper, Linda Torczon, in Engineering a Compiler (Second Edition), 2012*

### Accounting for Execution Frequencies

---

To account for the different execution frequencies of the basic blocks in the control-flow graph, the compiler should annotate each block with an estimated execution count. The compiler can derive these estimates from profile data or from heuristics. Many compilers simply assume that each loop executes 10 times. This assumption assigns a weight of 10 to a load inside one loop, 100 to a load inside two nested loops, and so on. An unpredictable if-then-else would decrease the estimated frequency by half. In practice, these estimates ensure a bias toward spilling in outer loops rather than inner loops.

To estimate the cost of spilling a single reference, the allocator adds the cost of the address computation to the cost of the memory operation and multiplies that sum by the estimated execution frequency of the reference. For each live range, it sums the costs of the individual references. This requires a pass over all the blocks in the code. The allocator can precompute these costs for all live ranges, or it can wait to compute them until it discovers that it must spill at least one value.

[View chapter](#)[Purchase book](#)

## Advances in Computers

---

*Neil Walkinshaw, in Advances in Computers, 2013*

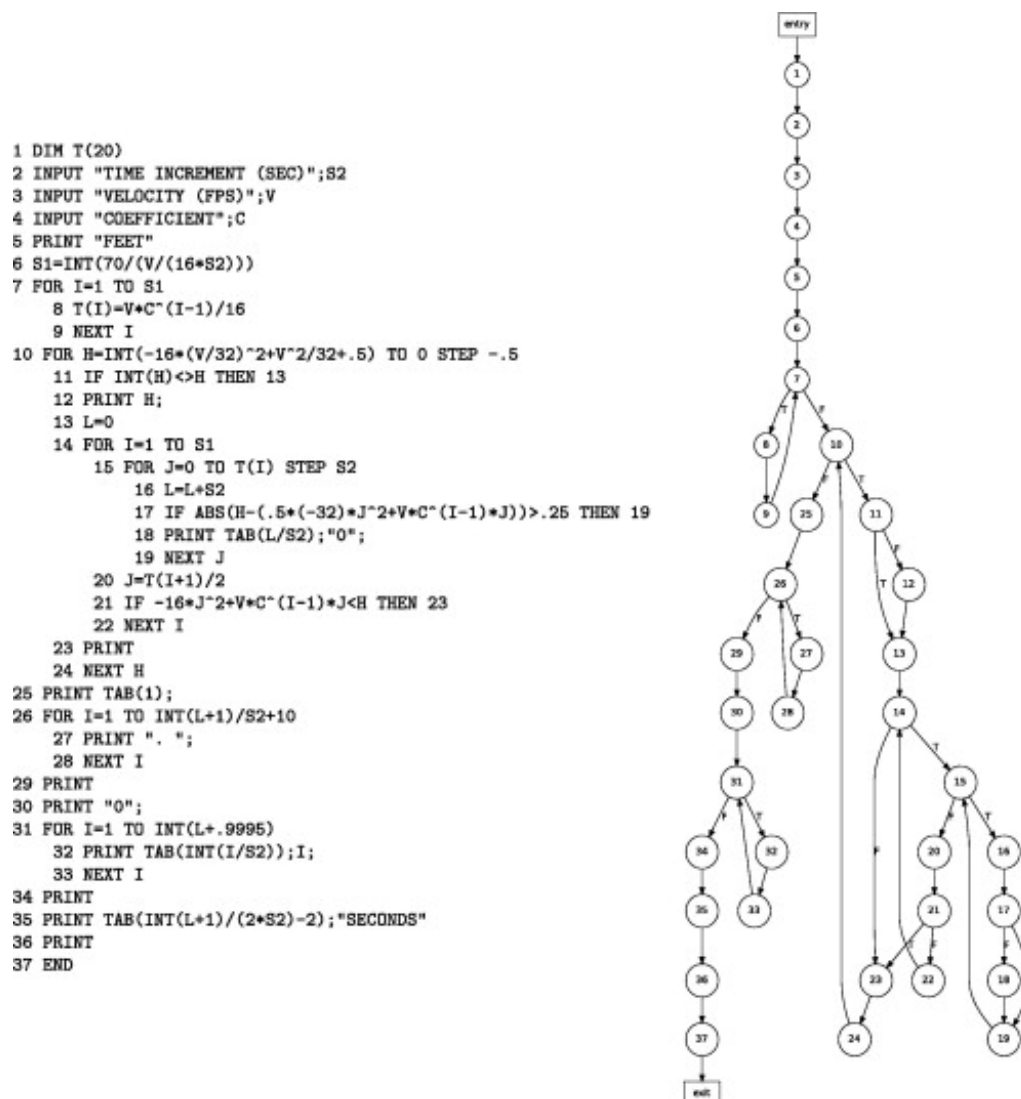
### 3.2.1 The Control Flow Graph

---

Individual procedures or routines in the source code can be represented as a *control-flow* graph (CFG) [2]. This is a directed graph, where nodes correspond to individual instructions, and edges represent the flow of control from one instruction to the next. Each graph has an entry point (indicating the point where the program starts), predicates such as if and while statements correspond to branches, and return statements or other terminating points in the program correspond to the exit nodes of the graph. Every possible path through the graph corresponds to a potential program execution (where loops in the graph indicate a potentially infinite number of executions).

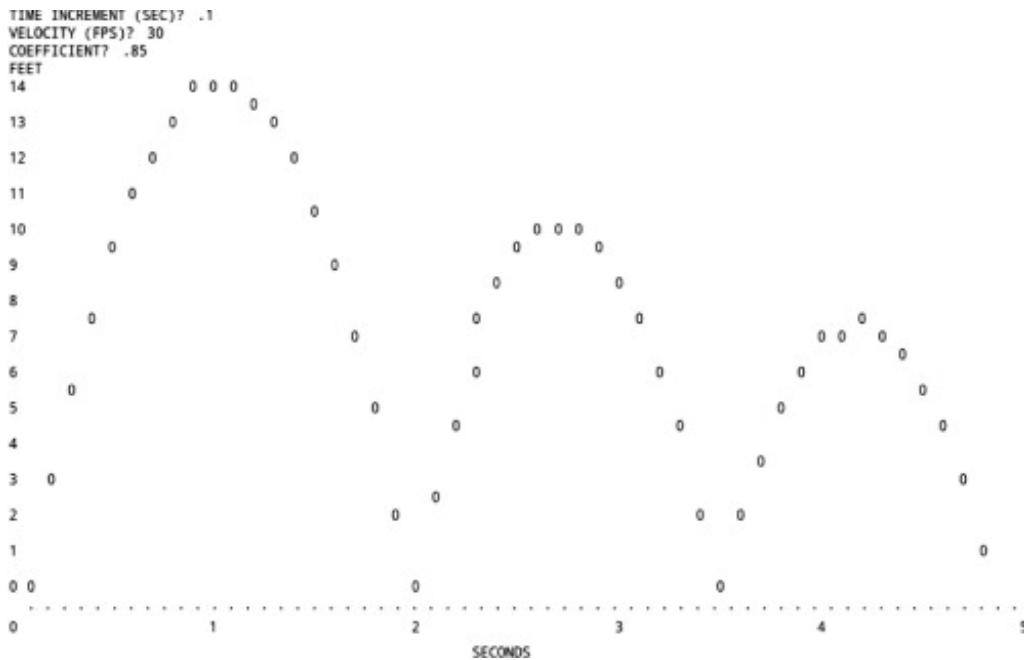
It is important to note the generality of this representation. CFGs can be used to represent control flow for programs written in any programming language. Whether the program in question is written in BASIC, Java, or C, or is the Assembly code post-compilation, it will contain decision points and instructions that are executed according to some predefined sequence, which can always be be represented in terms of a CFG.

To illustrate how a CFG is constructed we build a CFG for a simple BASIC routine (written by David Ahl [1]<sup>2</sup>). The code, shown on the left in Fig. 6, plots out the trajectory of the bounce of a ball (shown in Fig. 7), given a set of inputs denoting time-increment, velocity, and a coefficient representing the elasticity of the ball.



[Sign in to download full-size image](#)

Fig. 6. Bounce.bas BASIC routine to plot the bounce of a ball, and the corresponding CFG.



[Sign in to download full-size image](#)

Fig. 7. Output from bounce.bas.

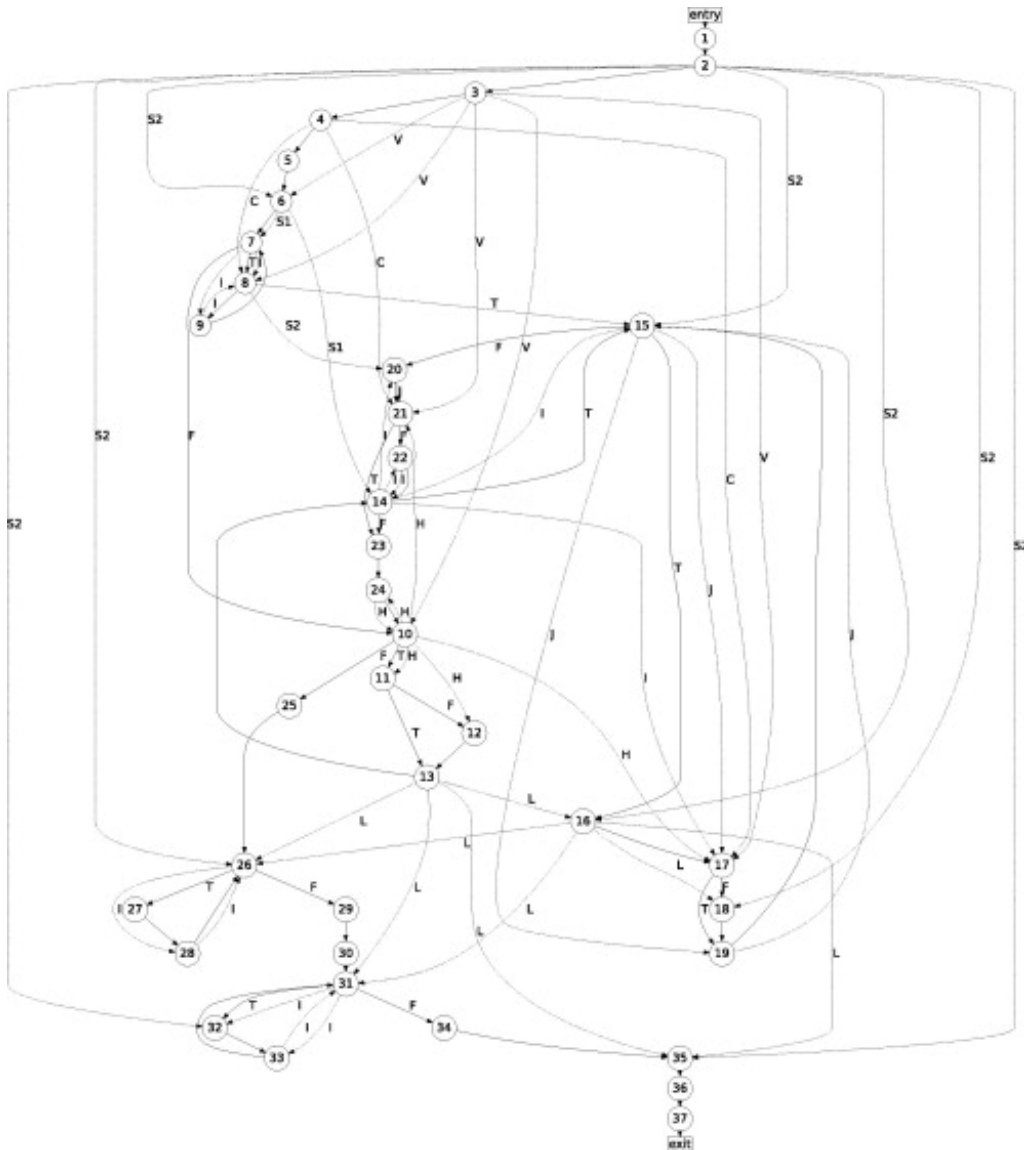
The CFG is shown on the right of the figure. It plots the possible paths through the function (the possible sequences in which statements can be executed). It starts and ends with the *entry* and *exit* nodes, which do not correspond to actual statements, but merely serve to indicate the entry and exit points for the function. Predicate-statements (e.g., if or while statements) are represented by branches in the CFG, where the outgoing edges denote the true or false evaluation of the predicate. Statements at the end of a loop include an edge back to the predicate.

### 3.2.1.1 Data Flow

The CFG in itself only conveys information about the possible order(s) in which statements can be executed. To add information about the possible flow of variable values from one statement to the other, we start by defining two types of relationship. For a given node  $s$  in a CFG, the function  $\text{def}(s)$  identifies the set of variables that are assigned a value (i.e., defined) at  $s$ . The function  $\text{use}(s)$  identifies the set of variables that are used at  $s$ .

The above  $\text{def}$  and  $\text{use}$  relations can, along with the CFG, be used to compute the *reaching definitions* [2]. For a given definition of a variable in the code, the reaching definitions analysis computes the subsequent points in the code where that specific definition is used. To make it slightly more formal, each node  $s$  in the CFG is annotated with a set of variable-node pairs  $\langle v, n \rangle$  for each  $v \in \text{use}(s)$ . These are

computed such that the value of  $v$  is defined at  $n$ , and there is a definition-free path with respect to  $v$  from  $n$  to  $s$ . These reaching definitions are visualized as dashed edges in the extended CFG, shown in Fig. 8.



[Sign in to download full-size image](#)

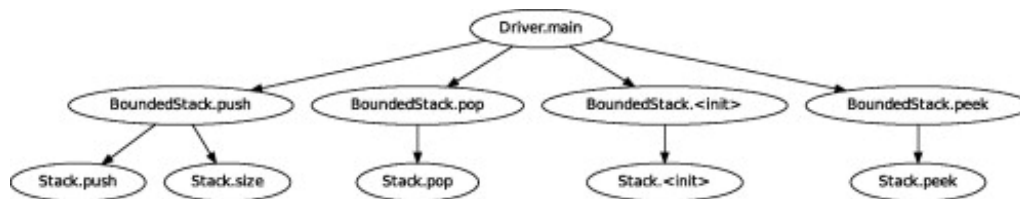
Fig. 8. CFG with reaching definitions shown as dashed lines.

Intuitively, this graph captures the basic elements of program behavior. The complex interrelationships between data and control are summarized in a single graph. The intuitive value is clear; it is straightforward to visually trace how different variables affect each other, and how different sets of variables feature in particular paths through the code. In practice such graphs are rarely used as visual aids, but form the basis for more advanced analyses, as will be shown below.

### 3.2.1.2 Call Graphs

The above representations are solely concerned with the representation of individual procedures or functions. In practice, for programs that exceed the complexity of the bounce program, behavior is usually a product of interactions between multiple functions. The possible calls from one method or function to another can be represented as a “call graph” [47]. Such graphs can, when used in conjunction with individual function CFGs, be used to compute data and control-flow relationships that span multiple procedures.

For this chapter an intuitive description of a call graph will suffice. A standard (context-insensitive) call graph consists of nodes that represent call statements within methods or functions, and edges represent possible calls between them. An example of a call graph with respect to the BoundedStack code is shown in Fig. 9. Due to the simplicity of the code, this call graph is straightforward; it is tree-shaped, there are no loops, and each call has only one possible destination.



[Sign in to download full-size image](#)

Fig. 9. Call graph for code in Fig. 1.

For larger systems, call graph computation can be exceptionally challenging [18], and has been the subject of an extensive amount of research. Calls in the source code do not necessarily have a unique, obvious target. For example, the target of a call in object-oriented system (because of mechanisms such as polymorphism and runtime binding) is often only decided at execution-time. The *pointer-analysis* algorithms that underpin these computations [18] lie beyond the scope of this chapter, and the possible problems of inaccuracy will be discussed in the wider context of static analysis in Section 3.5.

[View chapter](#)[Purchase book](#)

## Debugging Heuristics

*Robert Charles Metzger, in Debugging by Thinking, 2004*

### 8.9 Draw a diagram

Drawing the following diagrams can be a useful heuristic:

■

A control-flow graph with decisions actually taken



A data-flow graph that shows paths actually taken



Complex data structures that use pointers

A **control-flow graph** is a directed graph in which executed statements (or procedures) are represented by the nodes, and control flow is represented by the arcs. For the purpose of hypothesis generation, limit your control-flow graph to statements or procedures that were actually executed. You can develop this information using a program trace or an execution profile.

A **data-flow graph** is a directed graph in which assignments and references to variables are represented by the nodes, and information flow is represented by the arcs. For the purpose of hypothesis generation, limit your data-flow graph to assignments and references that were actually executed.

Obviously, it's much easier to develop this graph if you have first developed a statement-level control-flow graph.

What kind of hypotheses do these diagrams suggest? If you see an unanticipated control-flow path, a reasonable hypothesis is that there is a problem with one of the predicates that controls the path. If you see a missing flow of information, a reasonable hypothesis is that there is a problem with the predicates controlling the statements that did not get executed, in between the assignment and reference of the variable in question.

Given a diagram of the static relationships of a complex structure built from pointers, and given that you know that this structure is being manipulated when the problem manifests itself, ask yourself how this structure could be violated. What assignments could be missed or performed in the wrong order, causing the structure to lose integrity? Each would give rise to a reasonable hypothesis.

Other useful diagrams you may want to draw are application specific. You can work either from a representation of the actual behavior of the program, such as the control-flow or data-flow graph, or from a representation of the ideal behavior of the program, such as the data-structure layout. In the former case, look for actual anomalies. In the latter case, look for potential anomalies.

[View chapter](#)[Purchase book](#)

## Coverage-Based Software Testing

## 3.2 Statement Coverage and Basic Block Coverage

---

Control flow is a relation that describes the possible flow of execution in a program. A *control flow graph* (CFG) is a directed graph in which each node represents a statement and each edge represents the flow of control between statements within a function. That is, a CFG captures all paths that might be traversed during the execution of a function. A *SystemCFG* combines all the CFGs of a program by adding an edge to represent each function invocation.

The *statement coverage criterion* defines  $TR$  to include all the nodes in the SystemCFG. Thus, for  $T$  to satisfy *statement coverage*,  $T$  should execute every statement in the program at least once.

A *basic block* is a sequence of consecutive statements in which the flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end. CFGs and SystemCFGs are typically built based on basic blocks as opposed to statements. This is widely practiced because the resulting CFGs would be more compact (allowing for more efficient analyses), meanwhile preserving the same control flow information. Consequently, many testers choose to adopt *basic block coverage* as opposed to statement coverage.

Keep in mind that if a test suite  $T$  exhibits a coverage level of 100% for statement coverage, it will also exhibit 100% for basic block coverage (and vice versa). However, if the coverage level was less than 100% for statement coverage, say 90%, it will not necessarily be 90% for basic block coverage (and vice versa).

[View chapter](#)[Purchase book](#)

## Introduction to Optimization

---

Keith D. Cooper, Linda Torczon, in *Engineering a Compiler (Second Edition)*, 2012

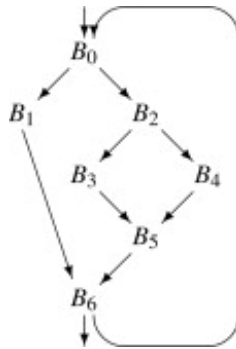
### Regional Methods

---

Regional methods operate over scopes larger than a single block but smaller than a full procedure. In the example control-flow graph (cfg) in the margin, the compiler might consider the entire loop,  $\{B_0, B_1, B_2, B_3, B_4, B_5, B_6\}$ , as a single region. In some cases, considering a subset of the code for the full procedure produces sharper analysis and better transformation results than would occur with information from the full procedure. For example, inside a loop nest, the compiler may be able to prove that a heavily used

pointer is invariant (single-valued), even though it is modified elsewhere in the procedure. Such knowledge can enable optimizations such as keeping in a register the value referenced through that pointer.

[Sign in to download full-size image](#)



The compiler can choose regions in many different ways. A region might be defined by some source-code control structure, such as a loop nest. The compiler might look at the subset of blocks in the region that form an *extended basic block* (ebb). The example cfg contains three ebbs:  $\{B_0, B_1, B_2, B_3, B_4\}$ ,  $\{B_5\}$ , and  $\{B_6\}$ . While the two single-block ebbs provide no advantage over a purely local view, the large ebb may offer opportunities for optimization (see Section 8.5.1). Finally, the compiler might consider a subset of the cfg defined by some graph-theoretic property, such as a *dominator relation* or one of the strongly connected components in the cfg.

## Extended Basic Block

a set of blocks  $\beta_1, \beta_2, \dots, \beta_n$  where  $\beta_1$  has multiple cfg predecessors and each other  $\beta_i$  has just one, which is some  $\beta_j$  in the set

## Dominator

In a CFG,  $x$  *dominates*  $y$  if and only if every path from the root to  $y$  includes  $x$ .

Regional methods have several strengths. Limiting the scope of a transformation to a region smaller than the entire procedure allows the compiler to focus its efforts on heavily executed regions—for example, the body of a loop typically executes much more frequently than the surrounding code. The compiler can apply different optimization strategies to distinct regions. Finally, the focus on a limited area in the code often allows the compiler to derive sharper information about program behavior which, in turn, exposes opportunities for improvement.

[View chapter](#)[Purchase book](#)

## Advances in Symbolic Execution



## 8.1 Finding Bugs and Improving Coverage

---

Batg [110] combines features of static analysis and bounded symbolic execution for test case generation. An initial static analysis identifies a list of potential errors on the control flow graph (CFG): potentially buggy nodes are identified, and the set of paths leading to these nodes is passed to the “path processor,” where symbolic execution is performed. All the paths with satisfiable path constraints are considered as real bugs, and one test case is generated for each of them; otherwise, these paths are unwound from its “CFG form” where the loops are considered as only one iteration, and the satisfiability of the new generated constraint is checked again. This process is done iteratively until a satisfiable context is found, which means a test case for the bug can be generated, or the unwinding bound is reached.

Following the guiding principle of RANDOOP [111], which relies on feedback-directed random input generation, Garg et al. [112] propose an automatic unit test generation technique to improve the coverage obtained by feedback-directed random test generation method. It employs dynamic symbolic execution on the generated test drivers and uses nonlinear solvers in a lazy manner to generate new test inputs for programs with numeric computations. It conducts analysis on unsatisfied cores returned by SMT solvers, checking whether a branch can be reached with another input or can be used to tell whether certain test suits cannot reach a target branch.

Bardin et al. [113] improve the coverage of generated test cases from a different perspective. They argue that although behaviors related to control are well handled, interesting behaviors related to data can still be missed in path-oriented criteria. Thus, they propose *label coverage*, a new coverage criterion based on *labels*, a well-defined and expressive specification mechanism for coverage criteria associated to program instructions. By appropriately changing the labeling function, multiple coverage criteria can be emulated such as instruction coverage, decision coverage, and condition coverage. A dynamic symbolic execution approach is developed for covering labels with no exponential blowup of the search space.

You et al. [114] propose an incremental test generation strategy using dynamic symbolic execution for generating test cases to satisfy the Observable Modified Condition/Decision Coverage (OMC/DC) [115]. OMC/DC is an improved coverage criterion based on Modified Condition/Decision Coverage (MC/DC), which is a condition-based criterion that is widely used in safety-critical systems. They use dynamic symbolic execution to generate tests that, first, satisfy a condition-based coverage criterion to exercise a particular part of the code (MC/DC) and, second, satisfy a dataflow criterion propagating the possibly corrupted state to a point where it is used by an observable output. A tag is assigned to each condition and

the propagation of tags to outputs approximates observability. Incremental test generation starts with concrete test inputs that satisfy an obligation and then invokes a model checker at each test step repeatedly to solve path conditions in an attempt to propagate tags through nonmasking paths toward outputs.

[View chapter](#)[Purchase book](#)