Tool Support for Improving Test Coverage

Susan Horwitz (horwitz@cs.wisc.edu)

University of Wisconsin and GrammaTech, Inc.

Abstract. Testing is a critical part of the software-engineering process. Coverage tools provide information about which components are exercised by a test suite, but they do not assist programmers with the important problem of how to increase coverage. We propose a tool to address that problem: Using the program's control and flow dependences, the tool helps programmers determine where to focus their efforts, and how to force a chosen component to be exercised.

1 Introduction

A common approach to testing software is the use of coverage criteria [5]; for example, using the all-statements criterion, a program is 100% covered if after running the program on all test inputs, every statement has been executed at least once. Tools such as Rational's PureCoverage [14], Bullseye Testing Technology's C-Cover [4], and Software Research, Inc's TCAT [15] help the programmer by measuring coverage: the tools instrument the program so that when it runs, a record is kept of which components executed. The tools can then report untested components to the programmer. However, these tools provide no help with the important problem of how to increase coverage.

Given a set of program components that have not yet been executed, the programmer must decide which component(s) to focus on next. The programmer would like to know, for each component:

- How easy will it be to find an input that causes the component to execute.
- How much benefit will there be in causing that component to execute; i.e., how many of the other currently untested components are also likely to be executed using the same, or similar inputs.

Furthermore, once the programmer has selected a component on which to concentrate, help is needed to determine how to force the execution of that component.

Example: Consider the program shown in Figure 1, which reads information about one employee, reads a print option, and then prints information either about the employee's average hours worked per day, or the current week's pay. (Apologies for the poor structure of the code – the use of the goto permits certain features of the proposed approach to be illustrated.) Comments on the right label the predicates, and line numbers are given on the left. Assume that the code has been tested on one input file, and that the untested statements are

```
(1)  read(jobCode, hourlyRate, hours);
(2)  read(printOption);
(3)  if (printOption == AV_HOURS) {                                   /* pred 1 */
(4)     print("av. hours per day =  ", hours/5.0);
(5)  }
(6)  else if (printOption == WEEKS_PAY) {                             /* pred 2 */
(7)            if (jobCode == SALARIED) {                             /* pred 3 */
(8)               pay = 40*hourlyRate;
(9)            }
(10)           else { if (hours > 40) {                               /* pred 4 */
(11)                      basePay = 40*hourlyRate;
(12)                      overtime = (hours-40)*1.5*hourlyRate;
(13)                      pay = basePay+overtime;
(14)                  }
(15)                  else {
(16)                      pay = hours*hourlyRate;
(17)                      goto L;
(18)                  }
(19)           }
(20)           if (hours > 60) {                                      /* pred 5 */
(21)              pay += BONUS;
(22)           }
(23)        L: print("weekly pay = ", pay);
(24)     }
(25) else print("unknown print option: ", printOption);
(26) print("all done!");
```

Fig. 1. An example program. This code has been tested on one input file, and the untested code is shown using bold font.

those shown using bold font. In this example, there are two blocks of untested code: The first (lines (6) - (24)) is the block that executes when predicate 1 evaluates to *false* and predicate 2 evaluates to *true* (i.e., when the input print option is `WEEKS_PAY`). The second (line (25)) is the single-line block that executes when both predicates 1 and 2 evaluate to *false* (i.e., when the input print option is neither `AV_HOURS` nor `WEEKS_PAY`). Within the first untested block, different code will execute depending on the values of predicates 3, 4, and 5. □

As mentioned above, the programmer would like help in choosing a key component of the program such that

- It will not be too difficult to figure out what input values will cause that component to execute.
- Many of the untested components in the program are likely to be executed using the same, or similar input values.

In this example, the same (small) amount of effort is required to force either of the outer two untested blocks to execute: in both cases, predicate 1 must evaluate to *false*; in the first case, predicate 2 must then evaluate to *true*, and in the second case, it must evaluate to *false*. However, the payoff in the first case is much greater: all but one of the untested components are in the block that is reached when predicate 1 is *false* and predicate 2 is *true*; only a single untested component is reached when both predicates are *false*. Given these observations, it is clear that it would be very useful to have a tool that:

– Guides the programmer to choose to concentrate on producing an input file that will cause predicate 2 to execute and to evaluate to *true*, and
– Assists the programmer in producing that file by indicating:
  - That in order to reach predicate 2, it is necessary first to reach predicate 1, and to have it evaluate to *false*, and
  - That the value of the variable `printOption` used in predicates 1 and 2 comes from the read statement at line (2) in the program.

A tool that has access to the program's control and flow dependences (defined in Section 2) can provide help with both of these problems, as described in sections 3, 4, and 5.

In the remainder of the paper, as in the example above, we assume that the criterion of interest is the all-statements criterion. However, it should be clear that the ideas presented apply equally well to other criteria, such as the all-branches criterion, that involve individual components (nodes or edges of the program's control-flow graph). Also, while our example program is written in C, the ideas presented here can be applied to any program for which control and flow dependences can be computed. Language features that make those computations difficult include interprocedural control flow (e.g., throw-catch or exceptions) parallelism (e.g., Java threads), and the use of standard libraries. Those issues are currently being addressed by a number of researchers, including [13], [10, 6, 11] and [12].

2 Background

2.1 Control Dependences

Intuitively, control dependences capture the fact that the number of times some program component will execute, or whether it will execute at all, may depend on the value of some predicate. For example, the statements in the *true* branch of an `if-then-else` statement will execute only if the condition evaluates to *true* (so those statements are said to be *true-control-dependent* on the `if` predicate), while the statements in the *false* branch will execute only if the condition evaluates to *false* (so those statements are said to be *false-control-dependent* on the `if` predicate).

Control dependences are defined in terms of the program's control-flow graph (CFG) [7]. For the purposes of defining control dependences, the `enter` node of a CFG is considered to be a predicate (that always evaluates to *true*) whose *true* control child in the CFG is the first statement in the program, and whose *false* control child in the CFG is a special `exit` node that follows the last statement in the program.

Definition 1. *A node n in a program's CFG is **v-control-dependent** on a node m iff n postdominates m's v CFG successor, but does not postdominate m (by definition, every node postdominates itself).*

Example: Figures 2 and 3 show the control-flow and control-dependence graphs for the program in Figure 1. Note that predicate 5 is *true*-control-dependent on both predicate 3 and predicate 4, because it postdominates both of their *true* successors in the CFG, but does not postdominate either of their *false* successors. Statement (23), `L: print("weekly pay = ...")` is not control dependent on predicates 3, 4, or 5, because it postdominates both CFG successors of each of those predicates (i.e., it executes whether they evaluate to *true* or to *false*). □

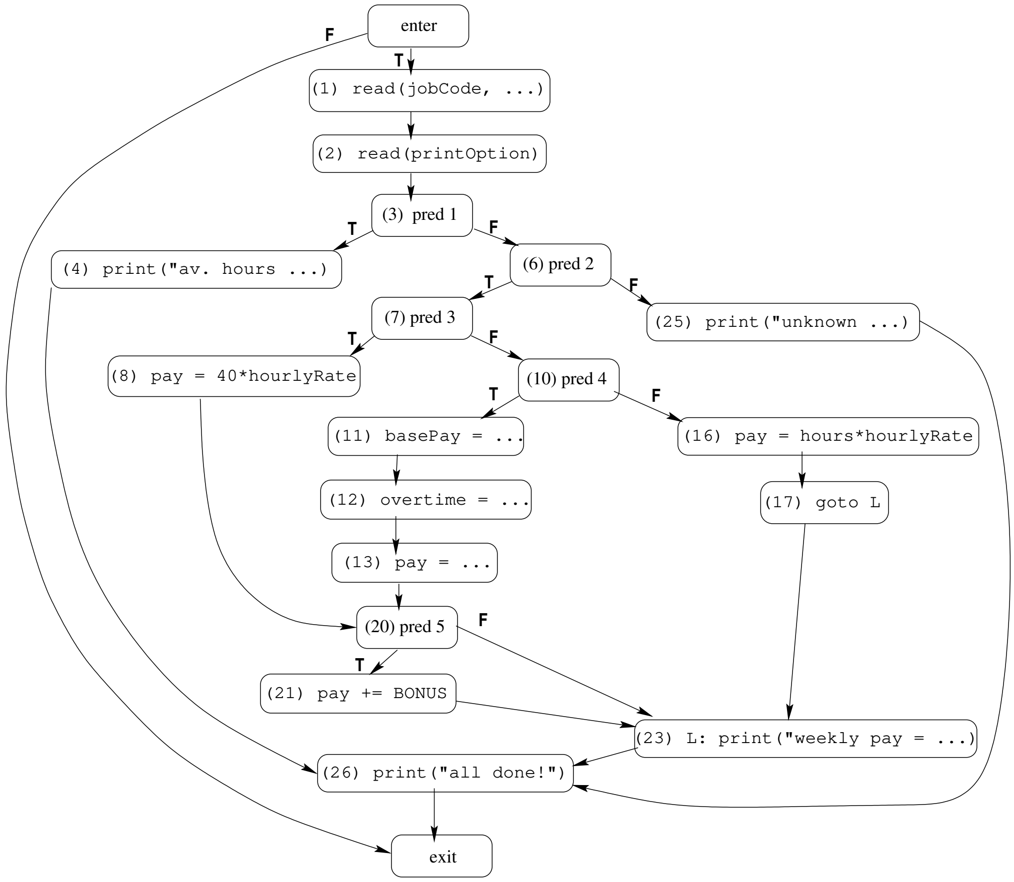

Fig. 2. The Control Flow Graph (CFG) for the example program.

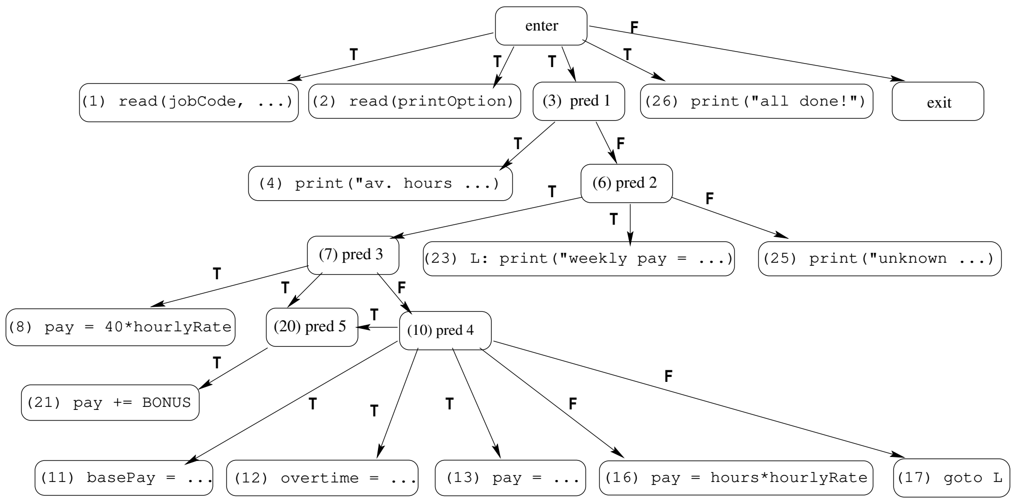

Fig. 3. The Control Dependence Graph for the example program.

2.2 Flow Dependences

Flow dependences are essentially *def-use* chains [1], representing the flow of values in the program.

Definition 2. *Node n is* ***flow dependent*** *on node m iff m defines a variable x, n uses x, and there is an x-definition-free path in the CFG from m to n.*

Example (refer to Figure 2): Statement (21), `pay += BONUS`, uses the variable `pay`. That variable is defined at three places in the program: statements (8), (13), and (16). There are `pay`-definition-free paths in the control-flow graph from (8) to (21), and from (13) to (21); therefore, statement (21) is flow dependent on statements (8) and (13). There is *no* path in the control-flow graph from (16) to (21); therefore, statement (21) is not flow dependent on statement (16). □

3 Choosing the Next Component on which to Focus

As discussed in the Introduction, when a programmer is trying to increase coverage, an important issue is which component to focus on next. In this section, we discuss using control dependences to address this issue. In particular, we describe how to compute three metrics that can be used to help answer the following two questions for each predicate in the program:

1. How easy will it be to find an input that causes the predicate to be executed?
2. How much benefit will there be in causing that predicate to be executed; i.e., how many of the currently untested components in the program will also be executed using the same, or similar inputs?

After defining the metrics, we first discuss the time required to compute them, then how they, together with additional information provided by the tool, can be used by the programmer to choose the next component on which to focus.

3.1 A Metric for the Ease of Forcing Execution of Component C

There may be many paths in a program's control-flow graph to a given component C (where a component is a statement or a predicate). C will execute if any one of those paths is followed. Following a particular path requires that the predicates along the path evaluate to the appropriate values. However, sometimes a predicate is actually irrelevant: if component C post-dominates a predicate in the control-flow graph, then C will execute regardless of the predicate's value.

Example: In the control-flow graph in Figure 2, all paths to statement (26), `print("all done!")`, include predicate 1; however, statement (26) will execute whether predicate 1 evaluates to *true* or to *false*, since it postdominates predicate 1 in the control-flow graph. Similarly, all paths to statement (23), `L: print("weekly pay = ", pay)` include predicates 1, 2, 3, and 4; however, since statement (23) post-dominates predicates 3 and 4, the values of those two predicates are not actually relevant to the execution of statement (23). □

An estimate of the effort needed to force an untested component C to execute can be computed by finding the path from a tested component to C that contains the fewest "relevant" predicates; i.e., we can define an *Ease-of-Execution* metric whose value is the number of relevant predicates on that path. The idea is that the programmer can reuse the part of the input that caused the component at the start of the path to execute, but then must figure out what the rest of the input must be to force execution to follow the path to C. Of course, this provides only a rough estimate of the actual effort needed to force C to execute, since in practice the predicates in a program are not independent, and it may be easier for the programmer to force a predicate to evaluate to one value than to another value. Nevertheless, we believe that this metric (when used with the other metrics defined in the next section) will be useful in helping the programmer to choose the next component on which to focus. In this context, absolute precision is not required, there just needs to be a reasonable correlation between the actual effort required and the values of the metrics for most components.

Example (refer to Figure 2): Consider statement (25), `print("unknown print option: ", printOption)`, in the example program. The Ease-of-Execution metric for this statement is 2, since it requires that both predicates 1 and 2 evaluate to *false*. The metric for statement (11), `basePay = 40*rate`, is 4, since it executes only when predicates 1 – 4 evaluate to *false*, *true*, *false*, *true*, respectively. Now consider predicate 5: `if (hours>60)`, at line (20). Note that there are two paths in the program's control-flow graph from the already-executed predicate 1 to this predicate: both start by following the *false* edge out of predicate 1, and the *true* edge out of predicate 2. One path then takes the *true* edge out of predicate 3, while the other takes the *false* edge out of predicate 3 and the *true* edge out of predicate 4. Thus, the first path includes 3 relevant predicates, while the second path includes 4 relevant predicates. Since the value of the metric is determined by the path with the fewest relevant predicates, the metric for predicate 5 is 3. □

This Ease-of-Execution metric can be computed efficiently for each component C using the program's control-dependence graph by finding the path in that graph from a tested predicate (or from the `enter` node) to C that includes the fewest predicates. In the control-dependence graph, each such path corresponds to one or more paths in the control-flow graph – the predicates on the control-dependence-graph path are the "relevant" predicates on the corresponding control-flow-graph paths (the possibility of irrelevant predicates is what can cause a single control-dependence-graph path to correspond to more than one control-flow-graph path).

Example: In the control-dependence graph in Figure 3 the path from predicate 1 to statement (25) includes two predicates (predicate 1 itself, and predicate 2); thus, as discussed above, the metric for that statement is 2. Similarly, the path to statement (11) contains four predicates, and there are two paths to predicate 5, one containing three predicates and the other containing four. The table in Figure 4 gives the Ease-of-Execution metric for each untested component in the example program. □

Untested Component	Ease Metric	Untested Component	Ease Metric	Untested Component	Ease Metric
line (6)	1	line (12)	4	line (21)	4
line (7)	2	line (13)	4	line (23)	2
line (8)	3	line (16)	4	line (25)	2
line (10)	3	line (17)	4		
line (11)	4	line (20)	3		

Fig. 4. The Ease-of-Execution metric for each untested component C in the example program: the number of predicates on the shortest path from a tested predicate (in this example, always predicate 1) to C in the program's control-dependence graph.

3.2 Two Metrics for the Benefit of Forcing Predicate P to Execute

Finding an input that forces a particular predicate P to execute and to evaluate to a specific value v can help improve test coverage in two ways:

1. There will be a set S of previously untested components that are guaranteed to execute whenever P evaluates to v, but that might not execute if P does not evaluate to v. Thus, forcing P to evaluate to v is guaranteed to increase coverage by at least the size of set S.
2. There will be another set T of previously untested components whose Ease-of-Execution metrics will decrease; thus, forcing P to execute and to evaluate to v will have "made progress" toward forcing the execution of each member of set T.

For instance, in our example program:

1. When predicate 2 evaluates to *true*, lines (7) and (23) are guaranteed to execute.
2. Predicate 2 evaluating to *true* lowers the Ease-of-Execution metric of statements (8), (10), (11), (12), (13), (16), (17), (20), and (21) by at least 2, possibly more, depending on the values of predicates 3, 4, and 5. Predicate 2 evaluating to *true* lowers the Ease-of-Execution metric of statement (25) by 1.

These observations suggest two useful metrics to report to the programmer for each predicate P, and each possible value v of P:

Must-Execute-Set Metric: The number of untested components that *must* execute if P executes and evaluates to v, and might not execute otherwise (the value of this metric is 2 for the pair (predicate 2, *true*) in the example program).

Improved-Ease-Set Metric: The total amount by which the ease metrics of untested components are guaranteed to be lowered if P executes and evaluates to v (the value of this metric is 19 for the pair (predicate 2, *true*) in the example program).

The two metrics can be computed efficiently using the control-dependence graph. Recall that component C is v-control-dependent on predicate P iff C postdominates P's v-successor (in the control-flow graph) but does not postdominate P. This means that if P evaluates to v, then C is guaranteed to execute, while if P evaluates to some other value, C may not execute. Thus, it seems that the Must-Execute-Set Metric for predicate P and value v could simply be the number of untested components that are v-control-dependent on P. However, there is a subtlety involving backedges in the control-flow graph. Such edges can cause a component C that dominates predicate P in the control-flow graph to be control dependent on P. For example, as illustrated in Figure 5, all of the statements in the body of a `do-while` loop dominate the loop predicate, and are also control dependent on it. If component C dominates predicate P, then whenever P executes, C will have already executed; i.e., C will be tested regardless of the value to which P executes, and therefore it would be wrong to include C when computing (P,v)'s Must-Execute-Set Metric (since that set is supposed to include only components that might *not* execute if P does not evaluate to v). Fortunately, this is an easy problem to solve: it simply requires that all backedges be removed from the control-flow graph before computing control dependences. (In the example of Figure 5, this would have the effect of removing all outgoing control-dependence edges from the `while` predicate.) Given control dependences computed using a control-flow graph with no backedges, the Must-Execute-Set Metric for each predicate P and value v is the number of untested components that are v-control-dependent on P.

The Improved-Ease-Set Metric can be computed for each predicate P and value v by determining, for each untested component C reachable in the control-dependence graph from P, how many predicates are on the shortest path from

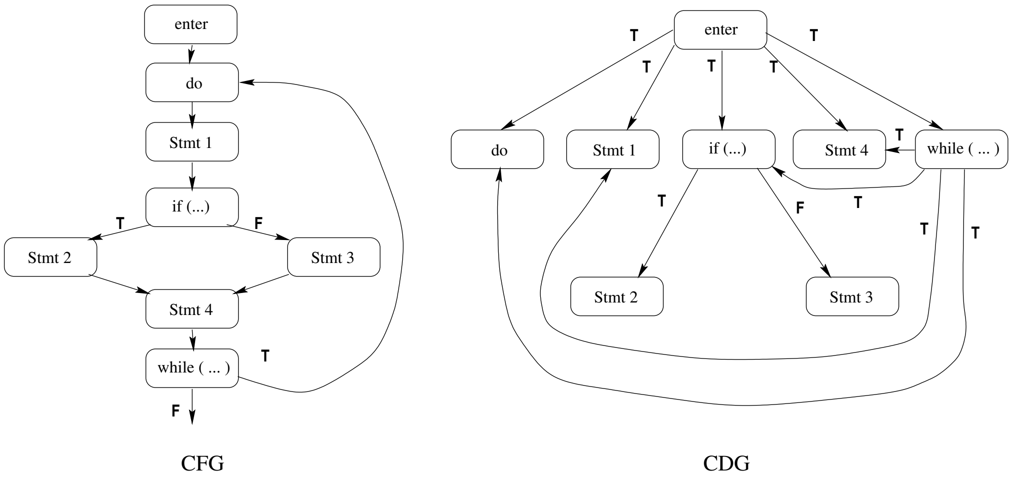

Fig. 5. Control-flow and control-dependence graphs for a `do-while` loop.

P to C (counting P itself when considering paths that start with a control-dependence edge labeled with a value other than v, and not counting P when considering paths that start with one of P's outgoing v edges). If the number of predicates is less than C's current Ease-of-Execution metric, then the difference in values is added to (P,v)'s Improved-Ease-Set Metric.

Note that we are proposing to supply the two "benefit" metrics only for predicates (rather than for all program components). It is true that forcing a statement S to execute also guarantees the execution of the other statements in the same weak region [2]. For example, forcing the statement on line (11) of our example program to execute guarantees that the statements on lines (12) and (13) will also execute. However, since only predicates have control-dependence successors, there is no obvious analog to the Improved-Ease-Set Metric for statements. Furthermore, in order to force a statement S to execute, the programmer must consider how to force the predicates that control S's execution to execute, and to evaluate to the appropriate values. Therefore, it seems reasonable for the programmer, and thus the tool, to concentrate on predicates.

Example: For each of the five predicates P in the example program, and for each value v to which P could evaluate, the table in Figure 6 lists the untested components that are guaranteed to execute if P evaluates to v (those that are v-control-dependent on P), those whose ease metrics are guaranteed to decrease if P evaluates to v (with the amount of decrease in parentheses), and the values of the two "benefit" metrics. □

3.3 Complexity

Ease-of-Execution Metric: The computation of the Ease-of-Execution metric for all untested components in the program can be performed using a single breadth-

Predicate	Value	Must Execute	Has Lowered Ease Metric	Metric 1	Metric 2
pred 1	*true*	- none -	- none -	0	0
	false	6	7(-1), 8(-1), 10(-1), 11(-1), 12(-1), 13(-1), 16(-1), 17(-1), 23(-1), 25(-1)	1	10
pred 2	*true*	6, 7, 23	8(-2), 10(-2), 11(-2), 12(-2), 13(-2), 16(-2), 17(-2), 20(-2)	3	16
	false	6, 25	- none -	2	0
pred 3	*true*	7, 8, 20	21(-3)	3	3
	false	7, 10	11(-3), 12(-3), 13(-3), 16(-3), 17(-3), 20(-2), 21(-2)	2	19
pred 4	*true*	10, 11, 12, 13, 20	21(-3)	5	3
	false	10, 16, 17	- none -	3	0
pred 5	*true*	20, 21	- none -	2	0
	false	20	- none -	1	0

Fig. 6. For each predicate in the example program, and for each possible value, the untested components that must execute, those whose ease metrics are guaranteed to be lowered (and by how much), and the values of the Must-Execute-Set Metric (Metric 1) and the Improved-Ease-Set Metric (Metric 2).

first search in the control-dependence graph. The queue is initialized to include all nodes that are control-dependence successors of some tested predicate, with metric = 1. Each time a node n is removed from the queue, each of its successors is considered. If a successor has not yet been visited, then its value is set to n's value + 1, and it is enqueued. The time required is linear in the size of the control-dependence graph.

Must-Execute-Set Metric: As stated in Section 3.2, the value of this metric for predicate P and value v is the number of untested components that are v-control-dependent on P. This number can be computed in time proportional to the number of components that are v-control-dependent on P simply by checking each one to see whether it has been tested.

Improved-Ease-Set Metric: The Improved-Ease-Set Metric for predicate P and value v can be computed by using breadth-first search to find the shortest path in the control-dependence graph from P to each untested component C, starting by putting all of P's v-successors on the queue. If the path to C is shorter than the current value of C's Ease-of-Execution Metric, then the difference in values is added to the running total for (P,v)'s Improved-Ease-Set Metric. In the worst case, this computation is linear in the size of the control-dependence graph.

Interprocedural Analysis To handle complete programs, control-dependence graphs are built for each procedure, and are connected by adding an edge from each node `call Proc` to `Proc`'s `enter` node. The techniques described above for computing the Ease-of-Execution and Improved-Ease-Set metrics work with a trivial extension in the interprocedural case: When during the breadth-first search a node `call Proc` is enqueued, if `Proc`'s `enter` node has not yet been visited, all of its successors should also be enqueued. In the case of the Ease-of-Execution metric, those nodes should also be given the same metric value as the call node. In the case of the Improved-Ease-Set metric, the length of the current path (counting the new interprocedural edges, as well as the edges out of the `enter` node as having length zero) should be compared to the nodes' recorded Ease-of-Execution metric value, updating (P,v)'s Improved-Ease-Set metric if appropriate.

When computing the interprocedural Must-Execute-Set metric for predicate P and value v, the call nodes that are v-control-dependent on P must be handled specially: for each such node, `call Proc`, the number of untested components that are *true*-control-dependent on `Proc`'s `enter` node must be included in the value of the metric. However, they should not be added multiple times when there are multiples calls to `Proc` that are v-control-dependent on P. A straightforward way to implement the computation of (P,v)'s Must-Execute-Set metric is to do a breadth-first search from P, initializing the queue to include all of P's v-control successors. The search continues only from call nodes (following their outgoing interprocedural edges) and from `enter` nodes (following their outgoing *true* edges). The final value of the metric is the number of untested components encountered during the search. The time required to compute the metric for (P,v) is proportional to the number of nodes that are v-control-dependent on P, plus the number of nodes that are *true*-control-dependent on the `enter` node for some procedure `Proc` called from a node that is v-control-dependent on P.

4 Using the Metrics

Our proposed testing tool will use the three metrics computed for each untested (predicate, value) pair to determine an ordering on those pairs, which will be reported to the programmer. One window will be used to display the program itself, and another window will be used to display the sorted list of pairs (with the values of the three metrics).

Experiments will be needed to determine the best way to compute the ordering (and, while a good default will be used, the programmer will also be given the opportunity to define alternative ways of combining the metric values to produce a sorted list). One reasonable possibility is to sort the pairs using the Ease-of-Execution metric as the primary key (sorting from low to high), the Must-Execute-Set metric as the secondary key (sorting from high to low), and the Improved-Ease-Set metric as the third key (sorting from high to low). The intuition is that predicates with low Ease-of-Execution metrics represent "low-hanging fruit"; i.e., it will be relatively easy for the programmer to figure out how to force those predicates to execute. By providing the two "benefit" metrics as well, the programmer can avoid wasting time focusing on those "easy" predicates that are unprofitable because their execution is not likely to cause very many untested components to execute.

Example: The information that would be provided to the programmer using this approach is shown in Figure 7. □

Predicate	Value	Ease-of-Execution Metric	Must-Execute-Set Metric	Improved-Ease-Set Metric
1: `if (printOption == AV_HOURS)`	*false*	0	1	10
2: `if (printOption == WEEKS_PAY)`	*true*	1	3	16
	false	1	2	0
3: `if (jobCode == SALARIED)`	*true*	2	3	3
	false	2	2	19
4: `if (hours > 40)`	*true*	3	5	3
	false	3	3	0
5: `if (hours > 60)`	*true*	3	2	0
	false	3	1	0

Fig. 7. The untested (predicate, value) pairs for the example program, sorted using the Ease-of-Execution metric as the primary key (sorted from low to high), the Must-Execute-Set metric as the secondary key (sorted from high to low), and the Improved-Ease-Set metric as the third key (sorted from high to low). The values of the three metrics are also shown.

The tool will also provide a way for the programmer to see the components that have contributed to the values of the three metrics: When the programmer chooses one of the predicate-value pairs (P, v) – by clicking on that line in the table – the tool will change the display of the program as follows:

- The text of the program is scrolled if necessary, so that the selected predicate is visible on the screen.
- The "relevant" predicates for P are displayed using red font. (If those predicates are not visible on the screen, a separate window is used to display the names of the files and procedures that contain relevant predicates.) Clicking on a relevant predicate causes the display of the value to which it must execute in order to follow the path to P.
- The components that are (interprocedurally) v-control-dependent on P are displayed using green font (again, a separate window is used to indicate where to find those components that are not on-screen).
- The components whose Ease-of-Execution metrics would decrease if P evaluates to v are displayed using blue font.

5 Help with Forcing the Execution of Predicate P

Once the programmer has selected a predicate-value pair (P, v) on which to focus, the next task is to consider what input values will cause P to execute and to evaluate to v. Our tool, which has access to the program's control and flow dependences can help in several ways.

5.1 The Reduced CFG

First, the tool can use the same information that was used to compute P's Ease-of-Execution metric to compute a reduced control-flow graph. Recall that the Ease-of-Execution metric was based on finding the shortest path in the control-dependence graph from an already-executed component C to P. Initially, the reduced control-flow graph includes only the paths that start from the beginning of the program, go through C, then reach P via a path in the CFG that corresponds to the control-dependence-graph path from C to P. The control and flow dependences induced by this reduced CFG are then computed and used to find all of the statements that involve computations that might affect which path is actually followed at runtime (i.e., the statements in the backward interprocedural slice from P [9]).[1] Irrelevant statements (those not in the slice) are removed, further reducing the size of the reduced CFG. The input statements in the reduced CFG are highlighted, using different colors for the ones that are relevant, and the ones that are irrelevant (an input statement is irrelevant if some input must be provided at that point, but the actual value can be arbitrary). Finally, the tool allows the user to "walk back" along the flow dependence edges in the reduced control-flow graph, to help illustrate the flow of values.

Example: Suppose that the programmer has decided to concentrate on forcing predicate 5 to evaluate to *true* (this is an unlikely choice, but is good for illustrative purposes). The Ease-of-Execution metric for predicate 5 was based on a control-dependence-graph path starting at (already executed) predicate 1. There is only one path in the CFG from the beginning of the program to predicate 1: nodes (1), (2), and (3). The control-dependence path from predicate 1 to predicate 5 goes via predicates 2 and 3. There is only one CFG path that corresponds to this control-dependence-graph path; that CFG path includes nodes (3), (6), (7), (8), and (20). Thus, the initial reduced CFG would contain just seven nodes: (1), (2), (3), (6), (7), (8), and (20). Since the statement at node (8), `pay = 40 * hourlyRate`, has no direct or transitive effect on any of the predicates, that node would also be eliminated. The code that corresponds to the final reduced control-flow graph is shown in Figure 8. Each predicate is annotated with the value to which it must evaluate in order to reach predicate 5; boxes are used to indicate the relevant inputs, and underlining is used to indicate the (single) irrelevant input. If the programmer asks to see the source of the value of variable `hours` used in predicate 5, the tool would display the input of that variable at line (1). Similarly, the programmer could ask to see the sources of the values of the variables used at the other relevant predicates. In this example, the information provided by the tool would quickly lead the programmer to understand that using `SALARIED` as the first input (for variable `jobCode`), an arbitrary value as the second input (for variable `hourlyRate`), a value greater than 60 for the third input (for variable `hours`), and `WEEKS_PAY` for

[1] We assume that input statements are chained together by control dependences as described in [3] so that every input statement in the reduced CFG is in the slice from P.

the final input (for variable `printOption`) would cause predicate 5 to evaluate to *true*. □

```
(1)   read(jobCode, hourlyRate, hours);
(2)   read(printOption);
(3)   if (printOption == AV_HOURS) {                /* pred 1: must be FALSE */

(5)   }
(6)   else if (printOption == WEEKS_PAY) {          /* pred 2: must be TRUE */
(7)              if (jobCode == SALARIED) {         /* pred 3: must be TRUE */

(9)              }

(20)             if (hours > 60) {                  /* pred 5 */
```

Fig. 8. The reduced version of the example program that is relevant to forcing predicate 5 to evaluate to *true*. The relevant inputs are indicated using boxes and the one irrelevant input is underlined.

5.2 Forbidden Predicate-Value Pairs

As discussed in Section 3.1, there may be multiple paths in the control-flow graph (and in the control-dependence graph) to a given predicate P. One example, caused by the use of a goto, was given in Section 3.1. However, non-structured code is not the only cause; for example, a predicate in a procedure that is called from multiple call sites will be reachable via multiple paths. While it is reasonable to start by considering the control-dependence path with the fewest predicates, the programmer may prefer to consider other paths as well. For example, the path used to compute the Ease-of-Execution metric for (P, v) might include a predicate-value pair (pred, val) such that the programmer cannot easily figure out how to force pred to evaluate to val. In this case, the programmer may wish to see a different reduced CFG that does not involve this problem pair. To permit this, our tool will allow the programmer to specify a set of prohibited predicate-value pairs; the tool will first find the best path in the control-dependence graph to P that does not include any prohibited pairs, and will then produce the corresponding reduced CFG. Computing the new path can be implemented easily and efficiently by using breadth-first search in the control-dependence graph from which edges that correspond to the forbidden pairs have been removed.

Example: Assume that the programmer has chosen to force predicate 5 to evaluate to *true*, but specifies that (predicate 3, true) is forbidden. The re-computation of the control-dependence-graph path would produce: (predicate 1, false)(predicate 2, true)(predicate 3, false)(predicate 4, true), because that is the shortest path in the control-dependence graph from predicate 1 to predicate 5 that does not include the forbidden predicate-value pair. The code that corresponds to the (final) reduced CFG that would be produced in this case is shown in Figure 9. □

```
(1)   read(jobCode, hourlyRate, hours);
(2)   read(printOption);
(3)   if (printOption == AV_HOURS) {                    /* pred 1: must be FALSE */

(5)   }
(6)   else if (printOption == WEEKS_PAY) {              /* pred 2: must be TRUE */
(7)             if (jobCode == SALARIED) {              /* pred 3: must be FALSE */

(9)             }
(10)            else { if (hours > 40) {                /* pred 4: must be TRUE */
(14)                   }
(19)            }
(20)            if (hours > 60) {                       /* pred 5 */
```

Fig. 9. The reduced version of the example program that is relevant to forcing predicate 5 to evaluate to *true*, given that the pair (predicate 3, true) is forbidden. The relevant inputs are indicated using boxes and the one irrelevant input is underlined.

6 Summary, Current Status, and Future Work

Testing is one of the most difficult aspects of software engineering. In practice, testers use tools that tell them what percentage of their code is covered by a given test suite, and their goal is to reach some level of coverage (typically something like 85%). While existing tools track code coverage, they do not provide help with the vital problem of how to increase coverage. We have proposed the design of a tool that does address this problem. The tool uses information about the program's control dependences to help the programmer to select (predicate, value) pairs such that it will not be too difficult to find an input that causes the selected predicate to evaluate to the chosen value, and such that forcing that evaluation will make good progress toward increasing code coverage. The tool then uses both control and flow dependences to help the programmer choose an input that will cause the selected predicate to execute, and to evaluate to the chosen value.

We have implemented an initial prototype of the tool using GrammaTech's CodeSurfer system [8]. CodeSurfer is an ideal platform on which to build since it provides many of the facilities needed to support our proposed tool:

– CodeSurfer processes C programs, computing their control and flow dependences.
– It provides an API that gives access to the program's CFG, control-dependence, and flow-dependence edges.
– It can import the program-coverage data gathered by PureCoverage [14], mapping per-line coverage information to per-CFG-node information.
– It provides high-level operations such as program slicing and chopping.

Future work includes completing the implementation of the tool and evaluating how well it works in practice, as well as investigating other uses for control and flow dependences in the context of testing. For example:

- How well does the Ease-of-Execution metric predict the actual effort required to force a given predicate to evaluate to a given value?
- How effective is the use of the reduced CFG in helping the programmer to write a test input that forces the chosen predicate to execute and to evaluate to the chosen value?
- Can control and flow dependences help with the problem of how to reuse existing test inputs when writing new ones?
- Can control and flow dependences help with the problem of automatic test-input generation?

Acknowledgements
This work was supported in part by the National Science Foundation under grants CCR-9970707, CCR-9987435 and DMI-0060607.

References

1. A. Aho, R. Sethi, and J. Ullman. Addison-Wesley, Reading, MA, 1986.
2. T. Ball. What's in a region - or - computing control dependence regions in linear time for reducible control-flow. *ACM Trans. on Programming Languages and Systems*, 2(1-4):1–16, 1993.
3. S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *Proc. ACM Symp. on Principles of Programming Languages (POPL)*, pages 384–396, January 1993.
4. Bullseye Testing Technology C-Cover. http://www.bullseye.com.
5. M. Davis and E. Weyuker. A formal notion of program-based test data adequacy. *Inform. and Contr.*, 56(1-2):52–71, Jan-Feb 1983.
6. M. Dwyer et al. Slicing multi-threaded Java programs: A case study. Technical Report 99-7, Kansas State University Computing and Information Sciences, 1999.
7. J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Trans. on Programming Languages and Systems*, 9(3):319–349, July 1987.
8. GrammaTech, Inc. Codesurfer user guide and reference manual, 2000.
9. S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. on Programming Languages and Systems*, 12(1):26–60, January 1990.
10. J. Krinke. Static slicing of threaded programs. In *Proc. ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Tools and Software Eng.*, June 1998.
11. M. Nanda and S. Ramesh. Slicing concurrent programs. In *Proc. Int. Symp. on Software Testing and Analysis*, August 2000.
12. A. Rountev and B. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. In *Int. Conf. on Compiler Construction*, April 2001.
13. S. Sinha, M. Harrold, and G. Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *Int. Conf. on Software Eng.*, pages 432–441, May 1999.
14. Rational Software. http://www.rational.com/products/pqc/index.jsp.
15. Software Research, Inc. http://www.soft.com/products/web/tcat.java.html.