Efficient Coverage Testing Using Global Dominator Graphs

Hira Agrawal
Telcordia Technologies*
445 South Street
Morristown, NJ 07960
hira@research.telcordia.com

Abstract

Coverage testing techniques, such as statement and decision coverage, play a significant role in improving the quality of software systems. Constructing a thorough set of tests that yield high coverage, however, is often a very tedious, time consuming task. In this paper we present a technique to find a small subset of a program's statements and decisions with the property that covering the subset implies covering the rest. We introduce the notion of a mega block which is a set of basic blocks spanning multiple procedures with the property that one basic block in it is executed iff every basic block in it is executed. We also present an algorithm to construct a data structure called the global dominator graph showing dominator relationships among mega blocks. A tester only needs to create test cases that are aimed at executing one basic block from each of the leaf nodes in this directed acyclic graph. Every other basic block in the program will automatically be covered by the same test set.

1 Introduction

The goal of coverage testing is to help software engineers construct good test sets that exercise their programs thoroughly [7, 19]. There are many coverage criteria, such as statement and decision coverage, that can be used to measure the goodness of a test set. Statement coverage criterion, sometimes also referred to as block or node coverage, requires that the program under test be run with enough inputs, or test cases, so that all its statements (or basic blocks, or flow graph nodes) get executed at least once. The motivation behind this criterion is that a statement must be exercised before a fault in it may be exposed. Decision coverage, similarly, requires that all decisions (or branches, or flow graph edges) must be exercised at least once. Developing a thorough set of test cases that yields high coverage, however, can be a very tedious, time consuming process.

We have previously developed a technique [1] to find a small subset of program's statements and decisions with the property that any test set that exercises all statements or decisions in the subset will exercise all statements or decisions in the program. That technique employed control flow analysis of individual functions or procedures in the program. In this paper, we present a stronger technique that also exploits the inter-procedural control flow relations to find even smaller subsets of program statements with the same property. The results can only improve because the sizes of the subsets identified by our intra-procedural technique provide upper bounds on the sizes of the subsets identified by our inter-procedural technique. We believe the latter technique will enable users to achieve higher coverages with significantly smaller test sets. Significantly smaller test sets usually translate into significant time- and cost savings.

We have also implemented the intra-procedural technique described in [1] in a coverage testing tool called χATAC, which is one of the tools in a comprehensive suite of software testing, debugging, understanding and maintenance tools, called χSuds [2]. We are currently in the process of implementing the inter-procedural technique described in this paper in χATAC.

In [1] we also reported an experiment involving eight software systems ranging from one to seventy five thousand lines of code where we fond that, on the average, test cases targeted to cover just 29% of the statements and 32% of the decisions identified using our technique

---

* Formerly Bellcore.

This paper appeared in the *Proceedings of the ACM SIGPLAN–SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'99)*, Toulouse, France, September 1999, pp. 11-20, and SIGSOFT Software Engineering Notes, vol. 24, no. 5, September 1999, pp. 11-20.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
©1999 ACM

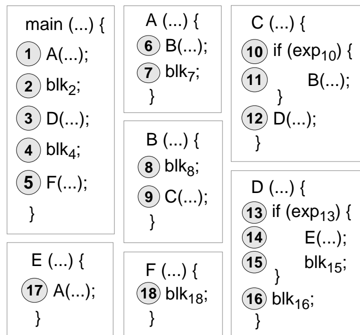

Figure 1: Example Program 1

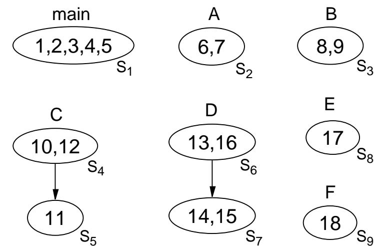

Figure 2: Super block dominator graphs of individual procedures in Example Program 1

ensured 100% statement and decision coverage, respectively. Moreover, we found that if the targeted statements and decisions were covered in the order prescribed by our technique, covering the first one third of them, on the average, implied covering over two thirds of all statements and decisions. We plan to repeat the same experiment with the new technique described here. We expect substantial reductions in the sizes of statement and decision subsets that need to be covered to ensure full coverage.

Besides saving the user time, as mentioned in [1], our techniques may also be used to reduce the space and time overhead imposed by coverage testing tools, by reducing the number of probes that need to be placed in the program. We need not instrument every basic block in the program. In the experiment mentioned above we found that, on the average, only 35% of the basic blocks in a program needed to be probed. If we know whether or not the probed basic blocks were covered, we could infer whether or not the remaining basic blocks were covered. We expect the inter-procedural technique described in this paper will improve this result significantly.

In the next section, we briefly review the intra-procedural, super block dominator graph algorithm we presented in [1] and motivate the need for computing global dominator graphs using an example. Then, in Section 3 we develop our algorithm to compute the global, inter-procedural dominator graph. In Section 4, we discuss how to compute correct global dominator graphs in the presence of interprocedural jump statements. We end with a discussion of related and future work in Section 5.

2 Background

In [1], we introduced the notion of a super block dominator graph. A super block, S, of a procedure, P, is a maximal set of basic blocks in P such that one basic block in S is executed *iff* every basic block in S is executed by any test case that causes P to terminate normally. A super block S_i is said to dominate another super block S_j if execution of the latter implies that of the former but not vice versa. The dominator relationship among the super blocks of a procedure can be nicely represented in the form of a directed acyclic graph, G, where S_i dominates S_j *iff* S_i is an ancestor of S_j in G. If we have a test set which causes all the leaves of a super block dominator graph to be executed, we can be sure that all super blocks, and hence all basic blocks in the procedure must be executed by the same test set. Thus testers only need to focus their attention towards covering one basic block from each leaf super block. Furthermore, the leaves may be weighted and ordered according to the total number of uncovered basic blocks among all the ancestor super blocks of a leaf. The exact same technique can be used to expedite decision coverage, if instead of basic blocks, decisions, or inter-block branches, are represented as nodes in a flow graph. The χATAC tool mentioned above, explicitly represents both basic blocks and decisions as nodes in the same flow graph and performs a combined dominator analysis involving both basic blocks and decisions.

We also presented a linear time algorithm in [1] (in terms of the flow graph size) to determine the super blocks of a procedure and construct its super block dominator graph. The algorithm essentially builds pre- and post-dominator trees of the procedure's flow graph, combines the two trees, and reduces the resulting graph by merging its strongly connected components and eliminating composite edges implied by dominator transitivity.

Figure 1 shows an example program with seven procedures, with all the details not relevant for our current purposes omitted (see [1] for a more comprehensive example). The statements labeled blk_i do not contain any branching constructs. The numbers in the shaded circles to the left of the code are included only for reference purposes. Figure 2 shows the super block dominator graphs of all seven procedures. The procedure named

```
build_global_dominator_graph (C)
  // Build the global dominator graph G of a program P with call graph C;
{ create a dummy function node, START;
  for each function, F, that can be invoked from outside of P, do
      add an edge (START, F) to C;
  T := build_annotated_loop_tree(C, START);
  G′ := build_loop_tree_dominator_graph(START);
  G″ := add_root_super_block_dominator_relations(G′);
  G := build_condensed_graph(G″);
  return G;
}
```

```
build_loop_tree_dominator_graph (R)
  // Build the global dominator graph, G_R, of the subtree rooted at R
  // in the annotated loop tree;
{
  for each child node, U, of R in the loop tree, do
      G_U := build_loop_tree_dominator_graph (U);
  if (type[R] = irreducible) then
      initialize G_R with a dummy node D;
      for each function F ∈ R do
          G_F := local_dominator_graph(F);
          add an edge (D, root_node(G_F)) to G_R;
  else
      G_R := local_dominator_graph(R);
  EntryFunctions[G_R] := Functions[R];
  sort the children of R topologically into a list, L, based on the annotated
      call edges among them;
  for each node U ∈ L, in order, do
      incorporate_dominator_graph (G_U, G_R);
  return G_R;
}
```

```
incorporate_dominator_graph (G_U, G_R)
  // Incorporate the global dominator graph G_U into G_R;
{
  S := ϕ;
  for each F ∈ EntryFunctions[G_U] do
      For each V ∈ CallSites[F] do
          if (V ∈ G_R) then S := S ∪ {V};
  W := nearest_common_dominator (S, G_R);
  add an edge (W, root_node(G_U)) to G_R;
}
```

```
add_root_super_block_dominator_relations (G)
  // Add dominator edges between all function call sites and the root
  // super blocks of the called functions to the global dominator graph G;
{
  for each function, F, do
      R := RootSuperBlock[F];
      for each V∈ CallSites[F] do
          add an edge (R, V) to G;
}
```

```
build_annotated_loop_tree (C, START)
  // Our adaptation of Havlak's algorithm to build the loop nesting
  // tree of a flow graph, for use with a call graph, C, rooted at
  // START; Our changes to the algorithm are shown in bold;
{
  Number the nodes of C using depth-first search from START,
      numbering in preorder from 1 to |N_C|;
  for w := 1 to |N_C| do
      nonBackPreds[w] := backPreds[w] := ϕ;
      header[w] := 1;
      type[w] := nonheader;
      for each edge (v, w) entering w in C do
          if (isAncestor(w,v)) then add v to backPreds[w];
          else add v to nonBackPreds[w];
  header[1] := nil;
  for w := |N_C| to 1 step -1 do
      P := ϕ;
      for each node v ∈ backPreds[w] do
          if (v ≠ w) then add Find(v) to P;
          else type[w] := self;
      worklist := P;
      if P ≠ ϕ then
          type[w] := reducible;
          LoopEntries[w] := {w};
      while worklist ≠ ϕ do
          select a node x ∈ worklist and delete it from worklist;
          for each node y ∈ nonBackPreds[x] do
              y′ := Find(y);
              if (not isAncestor(w, y′)) then
                  type[w] := irreducible;
                  add y′ to nonBackPreds[w];
                  Add x to LoopEntries[w];
              else if (y′ ∉ P) and (y′ ≠ w) then
                  add y′ to P and to worklist;
      for each node x ∈ P do
          header[x] := w;
          Union(x, w);
      for each node x ∈ P do
          for each annotated edge (x, y) in the loop tree do
              y′ := Find(y);
              if (y′ ≠ w) then
                  replace (x, y) with a new annotated edge (w, y);
      for each non-back edge (w, y) in C, do
          y′ := Find(y);
          if (y′ ≠ w) then
              add an annotated call edge (w, y′) to the loop tree;
  for each irreducible loop header, w, in a bottom up order, do
      for each x ∈ LoopEntries[w], x ≠ w, do
          delete all annotated edges incident upon x;
          merge the node containing x with w;
  return the annotated loop tree;
}
```

Figure 3: An algorithm to build global dominator graph of a program

D, for example, has two super blocks, s_6 and s_7. As s_7 is a leaf node, covering it would imply covering s_6. As s_7 contains two basic blocks, covering either one of them, say block 14, would imply covering the other. Thus the user only needs to create one test case aimed at covering block 14. The same test case will cover all other blocks in D—blocks 13, 15, and 16—automatically.

Among the nine super blocks shown in Figure 2, covering s_1 implies covering the most number of basic blocks, viz., five. The user should, therefore, construct a test case aimed at covering any one of the five basic blocks in main. That test case will automatically cover the other four basic blocks in main. Note, however, that three of these statements are calls to other procedures. Thus, at least some statements in the called procedures must also be covered by the same test case. But the super block dominator graph of main fails to capture this fact because it contains only intra-procedural control information—information that is strictly local to main.

In the next section we discuss how we can combine the super block dominator graphs of individual procedures and construct a graph that encodes inter-procedural control information as well. We call it the global dominator graph. To be consistent, we also refer to the super block dominator graphs of individual procedures as local dominator graphs.

Please note that the term dominator graph should not be confused with the term dominator tree. The former refers to dominator relationships among super blocks (in case of a local dominator graph) or mega

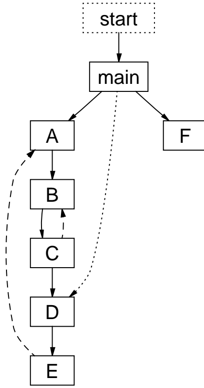

Figure 4: Call graph of Example Program 1 with its depth first search tree

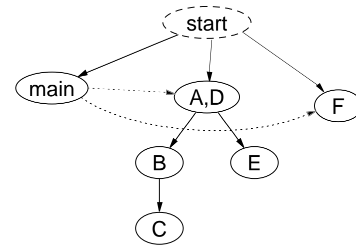

Figure 5: Annotated loop nesting tree of Example Program 1

blocks (in case of a global dominator graph) while the latter always refers to dominator relationships among a procedure's flow graph nodes. The latter is always a tree while the former may, in general, be a directed acyclic graph. Hence the qualifiers *graph* and *tree* should be used to differentiate between the two terms.

3 Global Dominator Graph

A test case that covers any one of the five statements in main, as we mentioned above, must cover some statements in other procedures as well. For example, it would cover blocks 13 and 16 in D because block 3 in main is a call to procedure D. The same holds true for blocks 6 and 7 in A and block 18 in F. Block 6 in A, however, is a call to procedure B. Thus covering it, in turn, would imply covering blocks 8 and 9 in B, and so on. It turns out that covering any one of the five blocks in main would imply covering fourteen of the eighteen blocks in the program. By the same reasoning, it can be deduced that covering blocks 14 or 15 in D would imply covering seventeen of the eighteen basic blocks in the program. We would, however, need to build a global dominator graph that captures inter-procedural dominator relationships to be able to deduce such inferences.

One way to construct a global dominator graph would be to build a global flow graph and apply a similar technique on it as the one we use to construct the local dominator graph of a procedure. But, then, we wouldn't be able to use the practical, linear or near linear dominator tree algorithms [10, 16] to construct the pre- and post-dominator trees of the global flow graph because of the presence of certain invalid, inter-procedural control paths in the global flow graph that fail to preserve the calling contexts of called procedures. It would, however, be easy to modify some of the simpler dominator tree algorithms [23, 3] to work with the global flow graph while still preserving the calling contexts of procedures, but those algorithms would have quadratic complexity in the size of the global flow graph which would make them impractical for use with real applications.

The approach we use, instead, is to first build the local dominator graphs of all procedures and then merge them using the call graph of the program. This would be straight forward if the call graphs were always acyclic graphs. If that were true, we could simply perform a topological sort of the call graph and construct the global dominator graph by sequentially merging the individual local dominator graphs in the order prescribed by the topological sort. To merge the local dominator graph of a procedure, P, in the global dominator graph, G, we would find all call sites of P in G, then find the nearest common dominator, A, of all those call sites, and attach the root node of the local dominator graph of P to A.

Call graphs, however, may be cyclic. They may, in fact, be irreducible graphs. To handle such graphs, we have adapted Havlak's algorithm to build a loop nesting tree of a flow graph [12] for use with call graphs. Havlak's algorithm, which build's upon Tarjan's algorithm for testing flow graph reducibility [26], essentially performs a depth first search of the graph to find all loop headers with incoming back edges and then examines the corresponding loops in reverse order to check if they have any "external" entry points besides the loop header, and simultaneously constructs the loop nesting tree in a bottom up manner. We have augmented this algorithm to add a second, orthogonal set of edges to the loop nesting tree reflecting the flow, or call, relationships among sibling loops or nonheader nodes. We then use this annotated loop tree to drive the order in which local dominator graphs of individual procedures are merged to build the global dominator graph.

Figure 3 shows our algorithm for building the global dominator graph including our adaptation of Havlak's

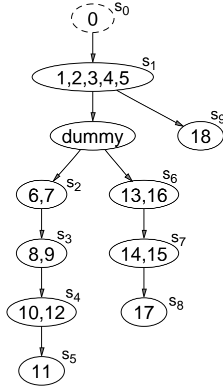

Figure 6: Initial global dominator graph of Example Program 1

procedure	call sites	root super-block
start		s_0
main	s_0	s_1
A	s_1, s_8	s_2
B	s_2, s_5	s_3
C	s_3	s_4
D	s_1, s_4	s_6
E	s_7	s_8
F	s_1	s_9

Figure 7: Call sites and root super blocks of procedures in Example Program 1

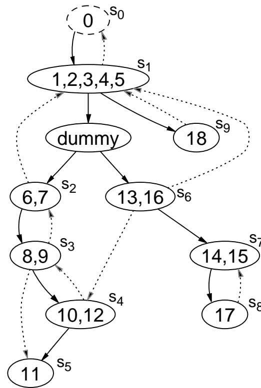

Figure 8: Intermediate global dominator graph of Example Program 1

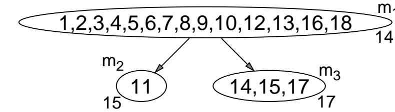

Figure 9: Final global dominator graph of Example Program 1

loop nesting tree algorithm. The statements in bold in the latter show our changes to his algorithm. The two for loops in bold towards the end of the large for loop add the annotated edges mentioned above to the loop nesting tree. The top level for loop in bold at the end of the algorithm merges nodes representing multiple loop entries of irreducible loops into single nodes. Unlike Havlak's algorithm, this step in our adaptation of the same ensures that we always end up with a unique annotated loop nesting tree irrespective of the depth first search tree chosen in the first step of the algorithm. For brevity, we have omitted details of certain straight forward functions in the algorithm such as build_condensed_graph, which merges the strongly connected regions of a graph into single nodes and eliminates any composite edges that are implied by transitive reduction of other edges from the graph [18].

Figure 4 shows the depth first search tree (in solid lines) along with back edges (in dashed lines) and forward- and cross edges (in dotted lines), for the call graph of Example Program 1. Figure 5 shows the corresponding annotated loop nesting tree, with tree edges shown in solid lines and the annotated edges shown in dotted lines. All internal nodes in the loop nesting tree represent loop headers (the start node is an exception and can be ignored for our purposes) while the leaves represent either self loops or nonheader nodes. Loop headers are further classified as being headers of reducible or irreducible loops. For a reducible loop, its header node lists its sole entry point. For an irreducible loop, it lists all of its entry points.

Figure 5 shows that there are two loops in the corresponding call graph: An irreducible loop that can be entered via A or D, and a reducible loop headed by B. The tree edge between the two loops implies that the latter is nested within the former. As there is no annotated edge between the two siblings, B and E, their dominator graphs may be incorporated in the dominator graph of their parent in any order. The two outgoing, annotated edges from main, however, require that its dominator graph be incorporated in the global dominator graph before those of its two siblings. Note that if a call graph had no cycles, there would be no internal nodes in its annotated loop tree (besides the start

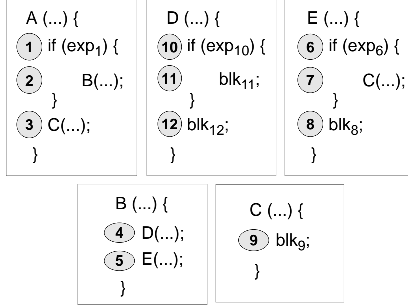

Figure 10: Example Program 2

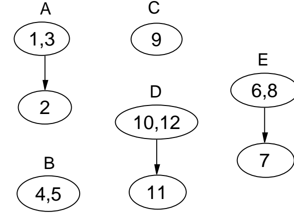

Figure 11: Super block dominator graphs of individual functions in Example Program 2

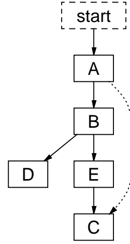

Figure 12: Call graph of Example Program 2 with its depth first search tree

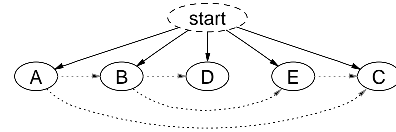

Figure 13: Annotated loop nesting tree of Example Program 2

node) and all procedures would become the children of the start node (see, e.g., the call graph in Figure 12 and its annotated loop tree in Figure 13). In that case, the annotated edges among them would represent the call graph itself, and the local dominator graphs of various procedures would be merged in the topological sort order of the call graph.

Figure 6 shows an intermediate step in the construction of the global dominator graph just after the local dominator graphs of all procedures have been merged in the order dictated by the annotated loop tree. Note that the global dominator graph at this stage is a directed acyclic graph. The next step is to add new edges to the graph that reflect the dominator relationships between call sites of procedures and their root super blocks. The root node of the local dominator graph of a procedure, P, contains its top level basic blocks that must all be executed every time P is invoked. In other words, the root super block of P dominates all call sites of P. Thus we add edges from the root super blocks of all procedures to all super blocks that contain a call to those procedures. Figure 7 lists the root super blocks of all procedures along with their call sites for Example Program 1.

Figure 8 shows the global dominator graph after the dominator edges from the root super blocks of all procedures to the corresponding call sites have been added to it. The resulting graph, as the figure shows, may contain cycles. All super blocks in a strongly connected region of this graph dominate each other. Thus any test case that executes one super block in it must execute all super blocks in it. We refer to these strongly connected regions as *mega blocks*. The final step in the construction of the global dominator graph is to find and merge all strongly connected components of this graph and delete any composite edges that can be induced by composition of other edges. As the nodes of the final global dominator graph represent mega blocks, we sometimes also refer to it as the mega block dominator graph. Figure 9 shows the final, global dominator graph of Example Program 1. Although it is a tree in this case, it is a directed acyclic graph in general (see, e.g., the global dominator graph in Figure 21). As our example global dominator graph has two leaves, the tester only needs to construct two test cases aimed at covering one basic block from each leaf. Those two tests will automatically cause all basic blocks in the program to be covered.

The nodes of a mega block dominator graph may be weighted according to the number of uncovered basic blocks among their ancestor nodes. Figure 9 shows the initial weights associated with all three mega blocks. They indicate that the tester should first try to cover a basic block in m_3, say block 15. Covering it will ensure covering seventeen of the eighteen basic blocks in the program (we have omitted the dummy start block). Next, covering block 11 will result in full block coverage.

Figure 10 shows another example program and Figure 11 shows the local, super block dominator graphs of all its procedures. Figure 12 shows the correspond-

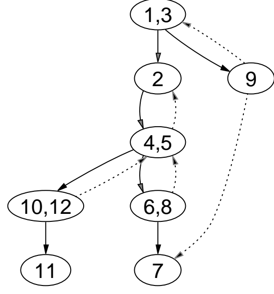

Figure 14: Intermediate global dominator graph of Example Program 2

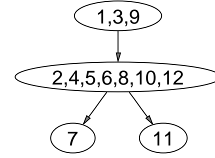

Figure 15: Final global dominator graph of Example Program 2

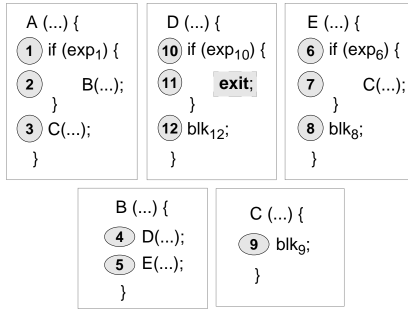

Figure 16: Example Program 3

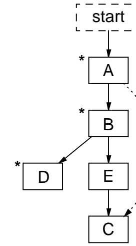

Figure 17: Call graph of Example Program 3 with its depth first search tree

ing call graph along with its depth-first-search tree and Figure 13 shows its annotated loop nesting tree. Note that all procedures of this program constitute the leaves of its loop nesting tree as its call graph is acyclic. The annotated edges in its loop nesting tree, therefore, represent the call graph itself. Figure 14 shows a step in the construction of the corresponding global dominator graph just after the dominator edges from root super blocks of all procedures to their corresponding call sites have been added and Figure 15 shows the final, global dominator graph after the strongly connected regions of the graph in Figure 14 have been merged and the redundant edges removed from the resulting graph. Note that the global dominator graph has only two leaves with basic blocks 7 and 11, both with the initial weight of 11. Thus designing a test case that covers either of the two leaf basic blocks would cover eleven of the twelve basic blocks in the program. A second test case covering the other, remaining leaf would ensure 100% block coverage of the program.

Although we have used the block coverage criterion in our examples above, the same algorithm works for decision coverage as well. In fact, we can perform a combined analysis taking both blocks and decisions into account at once.

Preliminary analysis of our algorithm indicates that the time it takes to complete is dominated by the time spent in the incorporate_dominator_graph procedure. The rest of the steps can be easily accomplished in time linear in the size of the annotated loop nesting tree. The time spent in the incorporate_dominator_graph procedure is, in turn, dominated by the time spent in calls to the nearest_common_dominator function. The latter determines the nearest common ancestor, or dominator, of a given set of nodes in a partially constructed global dominator graph. A simple, brute force algorithm to do the same would take $O(E+N.log(N))$ time where N and E denote the number of nodes and edges in the global dominator graph, respectively. As this function may be invoked at most p times, where p denotes the number of procedures in the program, the time complexity of our algorithm would, at worst, be $O(p.(E+N.log(N)))$. We suspect, however, that a much more efficient algorithm for finding the nearest common dominator of two or more nodes in a directed acyclic graph exists which would substantially improve the worst case complexity of our algorithm.

4 Handling Inter-procedural Jump Statements

Neither of the two examples we presented above contained any exit or halt statements available in most procedural programming languages. These statements interrupt the normal backtracking of control up the pro-

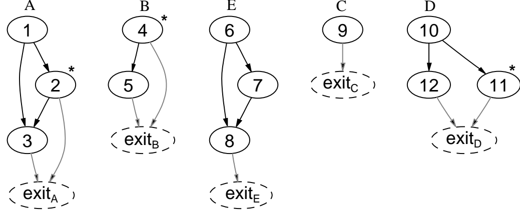

Figure 18: Annotated flow graphs of the procedures in Example Program 3

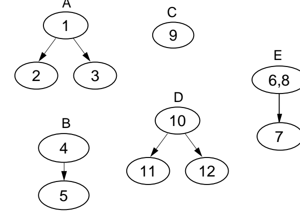

Figure 19: Super block dominator graphs of individual procedures in Example Program 3

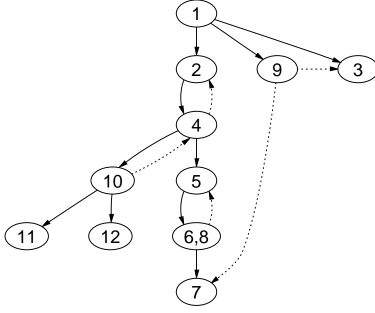

Figure 20: Intermediate global dominator graph of Example Program 3

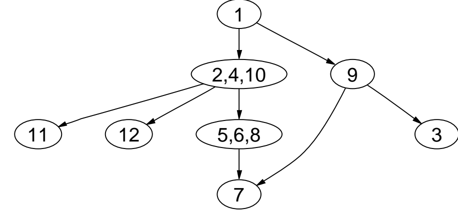

Figure 21: Final global dominator graph of Example Program 3

cedure call stack, and hence affect the computation of super and mega blocks as well as computation of dominator relationships among them. Consider, for example, Example Program 3 in Figure 16, which is identical to Example Program 2 in Figure 10 except that block 11 in procedure D has been replaced with an exit statement. In Example Program 2, blocks 2, 4, 5, 6, 8, 10, and 12 belonged to the same mega block (as shown in the global dominator graph in Figure 15) because execution of any one of these blocks implies that of all others. The same, however, does not hold true for Example Program 3 because of the presence of the exit statement in procedure D. Execution of block 4 no longer implies that of block 5 because the procedure invoked by block 4, viz., D, may not always return to its call site. Furthermore, the exit statement in D not only affects D alone but it affects all its ancestors in the call graph, viz., B and A, as well. Calls to B and A do not always return to their call sites either because of the exit statement in D. To denote this fact, we mark all procedures that contain an exit statement, as well as all their ancestors, with an asterisk in the call graph of the program. Figure 17 shows the marked call graph of Example Program 3.

If a node in a flow graph contains a call to a marked procedure then we know that control may not necessarily pass from this node to its successor node; an exit statement in the called procedure or one of its descendents may prevent the control from ever reaching the call site's successor node. This, in turn, implies that a node representing a call site of a marked procedure and its successor node may never belong the same super block. To denote this fact, we mark all call sites of all marked procedures with an asterisk in all procedure flow graphs. We further augment each procedure flow graph by adding a new, dummy exit node to it and creating edges from all marked call site nodes to the dummy exit node signifying the fact that control may be hijacked by an exit statement during the execution of the called procedure. We complete our augmentation of a procedure flow graph by adding edges from its normal return nodes to the dummy exit node signifying normal procedure returns. Figure 18 shows the augmented flow graphs of all procedures in Example Program 3. The dotted edge from node 2 to the dummy exit node of procedure A, for example, indicates that control does not always flow to node 3 after it has reached node 2 because the procedure called at node 2 does not always return.

Once we have created the augmented flow graphs of all procedures, we can use the normal, local dominator graph algorithm [1] on them to compute their super block dominator graphs. Then we can use the algorithm presented in Section 3 above to merge these super block dominator graphs and obtain the global dominator graph of the program. Figure 19 shows the super block dominator graphs obtained using the augmented flow graphs in Figure 18. Note how the super block dominator graphs of procedures A, B, and D differ from the corresponding super block dominator graphs of Example Program 2 in Figure 11. Figures 20 and 21 show

the intermediate and final steps in construction of the global dominator graph of Example Program 3, respectively. Note how the global dominator graph of Example Program 3 in Figure 21 differs from the global dominator graph of Example Program 2 in Figure 15. Block 12, for example, dominates block 11 in Example Program 2 but it is a leaf node in the global dominator graph of Example Program 3. Also the mega block containing the blocks 2, 4, 5, 6, 8, 10, and 12 for Example Program 2 has been split into three mega blocks for Example Program 2—one containing blocks 2, 4, and 10, one containing blocks 5, 6, and 8, and one containing block 12 by itself.

Other inter-procedural jump statements such as longjump calls in languages such as C and C++, and throw-catch exception handling constructs in languages such as Java and C++, may be handled in a similar fashion. With these statements we not only need to identify the procedures enclosing such statements but we also need to determine the procedures enclosing the target locations of those jump statements. Then, instead of marking all ancestors of a procedure that contains such a jump statement, we only need to mark its ancestors in the call graph up to, but not including, the procedure that contains the target location.

5 Related and Future Work

We are not aware of any other work in the literature that addresses the same problem we have addressed in this paper, viz. computation of dominator graphs at the inter-procedural level. At the intra-procedural level, we are only aware of one other work that addresses a similar problem: Bertolino and Marre's work on finding path covers in a flow graph [6]. Their "unconstrained arcs" are analogous to the leaves of our edge partion dominator graph in [1]. Our super blocks in [1] are also analogous to Ball's "weak regions" in [4] and Podgurski's control dependence regions arising from forward control dependences in [21].

Loyall and Mathisen present an algorithm to compute control dependences among procedures that may possibly contain exit statements [17]. Harrold, Rothermel and Sinha have developed algorithms to compute inter-procedural control dependences among flow graph nodes that also account for the presence of inter-procedural jump statements in programs [11, 25]. If we were to partition all nodes into equivalence classes based on inter-procedural control dependences obtained using their algorithms, we would end up with "strong" inter-procedural control regions analogous to the intra-procedural control regions obtained using algorithms such as those presented in [13] and [20]. Our mega blocks represent a much coarser partitioning of flow graph nodes compared to these inter-procedural strong control regions: Our one mega block consists of multiple such inter-procedural control regions.

The probe determination problem mentioned in Section 1 is a special case of the more general optimal profiling problem discussed in the literature [5, 8, 14, 22, 24]. Our techniques provide a relatively light weight solution for this special case compared to other techniques aimed at solving the general problem.

As mentioned earlier, our inter-procedural algorithm uses an adaptation of Havlak's loop nesting tree algorithm [12] which, in turn, is an adaptation of Tarjan's algorithm for testing flow graph reducibility [26].

The effectiveness of our inter-procedural dominator graph algorithm is a function of how accurate the call graph of the program is. When programs contain procedure variables, certain edges may be missing from the call graph that is input to our algorithm. This may result in misleading weights being assigned to basic blocks. This may not pose a big problem for coverage testing purposes. In the presence of inaccurate information, users may fail to achieve the level of coverage they would anticipate with individual tests. They may, at worst, have to develop more test cases to achieve the desired coverage. If the same techniques are used for efficient probing purposes, however, as mentioned in Section 1, presence of incomplete information in the input may lead to incorrect computation of coverage results. Algorithms to compute call graphs in the presence of procedure variables (see, e.g., [9, 15]) may offer partial solutions to this problem.

We are presently implementing the inter-procedural algorithm described in this paper in the χATAC tool mentioned in Section 1. Once this implementation is ready, we plan to conduct experiments evaluating its effectiveness along the same lines as the experiment we reported in [1]. Our algorithm based on the use of augmented loop nesting tree to drive merging of local dominator graphs to construct the global dominator graph may also be used as a framework to solve other inter-procedural control analysis problems. We also plan to investigate development of such a framework.

References

[1] H. Agrawal. Dominators, super blocks, and program coverage. In *Conference Record of the 21st Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL'94)*, pages 25–34. ACM Press, Jan. 1994.

[2] H. Agrawal, J. L. Alberi, J. R. Horgan, J. J. Li, S. L. London, W. E. Wong, S. Ghosh, and N. Wilde. Mining system tests to aid software

maintenance. *IEEE Computer*, 31(7):64–73, July 1998.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[4] T. Ball. What's in a region? or computing control dependence regions in near-linear time for reducible control flow. *ACM Letters on Programming Languages and Systems*, 2(1–4):1–16, Mar.–Dec. 1993.

[5] T. Ball and J. R. Larus. Optimally profiling and tracing programs. In *Conference Record of the Nineteenth ACM Symposium on Principles of Programming Languages (POPL'92)*, pages 59–70. ACM Press, Jan. 1992.

[6] A. Bertolino and M. Marre. Automatic generation of path covers based on the control flow analysis of computer programs. *IEEE Transactions on Software Engineering*, 20(12):885–899, Dec. 1994.

[7] R. A. DeMillo, W. M. McCracken, R. J. Martin, and J. F. Passafiume. *Software Testing and Evaluation.* Benjamin/Cummings Publishing Co., 1987.

[8] S. L. Graham, P. B. Kessler, and M. K. McKusick. An execution profiler for modular programs. *Software Practice and Experience*, 13:671–685, 1983.

[9] M. W. Hall and K. Kennedy. Efficient call graph analysis. *ACM Letters on Programming Languages and Systems*, 1(3):227–242, Sept. 1992.

[10] D. Harel. A linear time algorithm for finding dominators in flow graphs and related problems. In *Proceedings of the 17th ACM Symposium on Theory of Computing*, pages 185–194, May 1985.

[11] M. J. Harrold, G. Rothermel, and S. Sinha. Computation of interprocedural control dependence. In *Proceedings of the ACM International Symposium on Software Testing and Analysis*, pages 11–20. ACM Press, Mar. 1998.

[12] P. Havlak. Nesting of reducible and irreducible loops. *ACM Transactions on Programming Languages and Systems*, 19(4):557–567, July 1997.

[13] R. Johnson, D. Pearson, and K. Pingali. The program structure tree: Computing control regions in linear time. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI'94)*, pages 171–185. ACM Press, June 1994.

[14] D. E. Knuth and F. R. Stevenson. Optimal measurement points for program frequency counts. *BIT*, 13:313–322, 1973.

[15] A. Lakhotia. Constructing call multigraphs using dependence graphs. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 273–284. ACM Press, Jan. 1993.

[16] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

[17] J. Loyall and S. Mathisen. Using dependence analysis to support the software maintenance process. In *Proceedings of the IEEE Conference on Software Maintenance*, pages 282–291. IEEE Computer Society Press, Sept. 1993.

[18] D. M. Moyles and G. L. Thompson. An algorithm for finding a minium equivalent graph of a digraph. *Journal of the ACM*, 16(3):455–460, July 1969.

[19] G. J. Myers. *The Art of Software Testing.* John Wiley & Sons, 1979.

[20] K. Pingali and G. Bilardi. Optimal control dependence computation and the roman chariots problem. *ACM Transactions on Programming Languages and Systems*, 19(3):462–491, May 1997.

[21] A. Podgurski. Forward control dependence, chain equivalence, and their preservation by reordering transformations. Technical Report CES-91-18, Computer Engineering & Science Department, Case Western Reserve University, Cleveland, Ohio, U.S.A., 1991.

[22] R. L. Probert. Optimal insertion of software probes in well-delimited programs. *IEEE Transactions on Software Engineering*, SE-8(1):34–42, Jan. 1982.

[23] P. W. Purdom, Jr. and E. F. Moore. Immediate predominators in directed graphs. *Communications of the ACM*, 15(8):777–778, Aug. 1972.

[24] C. V. Ramamoorthy, K. H. Kim, and W. T. Chen. Optimal placement of software monitors aiding systematic testing. *IEEE Transactions on Software Engineering*, SE-1(4):403–411, Dec. 1975.

[25] S. Sinha, M. J. Harrold, and G. Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *Proceedings of the 21st International Conference on Software Engineering*, pages 432–441. IEEE Computer Society Press, May 1999.

[26] R. E. Tarjan. Testing flow graph reducibility. *Journal of Computer and System Sciences*, 9:355–365, 1974.