Proc. of the ACM Int'l. Symp. on Softw. Testing and Analysis, Boston, MA, June, 1993, pages 139-148

Efficient Construction of Program Dependence Graphs*

Mary Jean Harrold, Brian Malloy and Gregg Rothermel
Department of Computer Science
Clemson University
Clemson, SC 29634-1906

Abstract

We present a new technique for constructing a program dependence graph that contains a program's control flow, along with the usual control and data dependence information. Our algorithm constructs a program dependence graph while the program is being parsed. For programs containing only structured transfers of control, our algorithm does not require information provided by the control flow graph or post dominator tree and therefore obviates the construction of these auxiliary graphs. For programs containing explicit transfers of control, our algorithm adjusts the partial control dependence subgraph, constructed during the parse, to incorporate exact control dependence information. There are several advantages to our approach. For many programs, our algorithm may result in substantial savings in time and memory since our construction of the program dependence graph does not require the auxiliary graph. Furthermore, since we incorporate control and data flow as well as exact control dependence information into the program dependence graph, our graph has a wide range of applicability. We have implemented our algorithm by incorporating it into the Free Software Foundation's GNU C compiler; currently we are performing experiments that compare our technique with the traditional approach.

*This work was partially funded by NSF under grant CCR-9109531 to Clemson University.

1 Introduction

Structural testing techniques use intermediate representations of programs to select test data and determine test adequacy. A common program representation is the *control flow graph,* in which each node represents a program statement, and each edge represents a transfer of control between statements. Several recent testing techniques make use of *control dependence* information[1] that is not explicitly represented in a control flow graph. For example, control dependence information has been used to select data and determine adequacy[16], and to extend data flow testing techniques[7]. Control dependence information has also been used to generate reduced test sets for programs[9]. Moreover, several techniques used for regression testing [4, 5, 10] need control dependence information to identify the retesting required after changes are made to a program. Finally, both static and dynamic slicing techniques require control dependencies[1, 2, 17]. Thus, a program representation that contains explicit control dependence information is extremely useful for testing.

One program representation that encodes both control and data dependencies is the *program dependence graph* (PDG)[8]. Previous techniques for PDG construction[8, 6] rely on a control flow graph to identify control dependencies and compute data flow information. For programs with *goto* statements, control flow information is required to identify control dependencies and loops. However, some programming languages, such as Modula-2, enforce structured design since the language contains no *goto* statement. Even languages that contain *goto* statements, such as C and Ada, promote structured design. In an experiment using the UNIX utilities, we found that over 80 percent of the procedures contain no *goto* statements. Our research has shown that for these programs, control dependencies can be obtained without a control flow graph.

In previous work[11], we described a technique

[1] s_i is control dependent on s_j if the execution of s_i depends on the outcome of s_j.

that builds a control dependence subgraph during the parsing phase of compilation. That technique, however, is only applicable to structured programs. In this paper, we present a new technique for constructing a PDG that handles both structured and unstructured programs. Our algorithm constructs the PDG during the parse phase of compilation and the resulting PDG contains control flow information along with the usual control and data dependence information. For structured programs, the algorithm constructs the control dependence subgraph without using the control flow graph of the procedure. If structured transfers of control, such as *break*, *continue* and *exit*, are encountered, our algorithm can still identify the program's structure and construct the control dependence subgraph without requiring the entire control flow graph. If explicit *goto* statements are encountered, the resulting graph requires additional processing to obtain the exact control dependence subgraph.

A major part of our algorithm involves ordering the nodes in the control dependence subgraph to identify control flow in the program. Our ordered control dependence subgraph incorporates control flow either implicitly through node order or explicitly through the creation of control flow edges. To obtain the data dependence subgraph, we perform data flow analysis directly on the control dependence subgraph augmented with explicit control flow information where required. Using data flow sets, we add data dependence edges to get the data dependence subgraph and the PDG. We have incorporated our PDG construction algorithm into the GNU C compiler `gcc`[2][18], where we construct the PDG during the parse phase of compilation. In this implementation, we use the PDG as our only intermediate representation of the program.

There are several advantages to our approach. For many programs, our approach may result in substantial savings in the time and memory it takes to construct the PDG, since we eliminate construction and analysis of the control flow graph. However, our PDG can be used for all applications that require information on control flow, such as data flow analysis, test case generation, regression testing and dynamic execution profiling. Further, our PDG contains a program's control flow but retains the program's exact control dependencies, whereas previous techniques incorporated control flow but only approximated control dependencies. We have used our PDG to perform dynamic data flow analysis and applied it to data flow testing of parallelized code[12].

In the next section, we provide background about the PDG. In section 3, we present our technique for building the PDG by first discussing construction of the control dependence subgraph, and then presenting the data flow analysis algorithms used to build the data dependence subgraph. We discuss our implementation in section 4 and discuss some applications to testing in section 5. Finally, our conclusions are presented in section 6.

[2]Copyright (C) 1987, 1989 Free Software Foundation, Inc, 675 Mass Avenue, Cambridge, MA 02139.

2 Background

A PDG encodes control dependencies in a *control dependence subgraph* (CDS). To facilitate analysis and obtain the CDS, a control flow graph is often augmented with unique *entry* and *exit* nodes. Figure 1 gives a program segment and its control flow graph. For statements (nodes) X and Y in a control flow graph, if X is *control dependent* on Y then Y must have two exits where one of the exits from Y always causes X to be executed, and the other exit may result in X not being executed[8]. We say that X is control dependent on Y with the label (true or false) that definitely causes X to execute. A statement X may be control dependent on several statements in the program. Since these statements form nested sequences of control dependencies, we can always identify *immediate* control dependencies for X. For example, in the control flow graph of Figure 1, statement (node) S6 is control dependent on both P5-false and P3-false since both of these must hold for statement S6 to execute, but statement S6 is immediately control dependent only on P5-false.

The nodes in a CDS represent statements or regions of code that have common control dependencies. The CDS of the PDG for the program segment of Figure 1, constructed using the Ferrante, Ottenstein and Warren method[8], is shown on the right of Figure 1. The node numbers correspond to statement numbers in the program. A CDS contains several types of nodes. Statement nodes, shown as ellipses in Figure 1, represent statements in the program. Circles represent region nodes, which summarize the control dependencies for statements in the region. Predicate nodes, from which two edges may originate, are represented as squares in the figure. Predecessors of a node in the CDS are known as its *parent* regions, and successors of a region or predicate node are known as its *children*. Further, children with the same parents are known as *siblings* of each other in the CDS. Control dependencies are explicitly represented in the CDS. For example, it is clear from the CDS that statement (node) S6 is control dependent on both P5-false and P3-false, but immediately control dependent on P5-false.

Region nodes in the CDS summarize the control dependencies of a group of statements. For example, in Figure 1, both statement node S6 and predicate node P7 are control dependent on P5-false; without region node R3, there would be two edges labeled "F" from node P5.

A second subgraph of the PDG, the *data dependence subgraph* (DDS), encodes data dependencies among statements. The DDS is obtained by creating

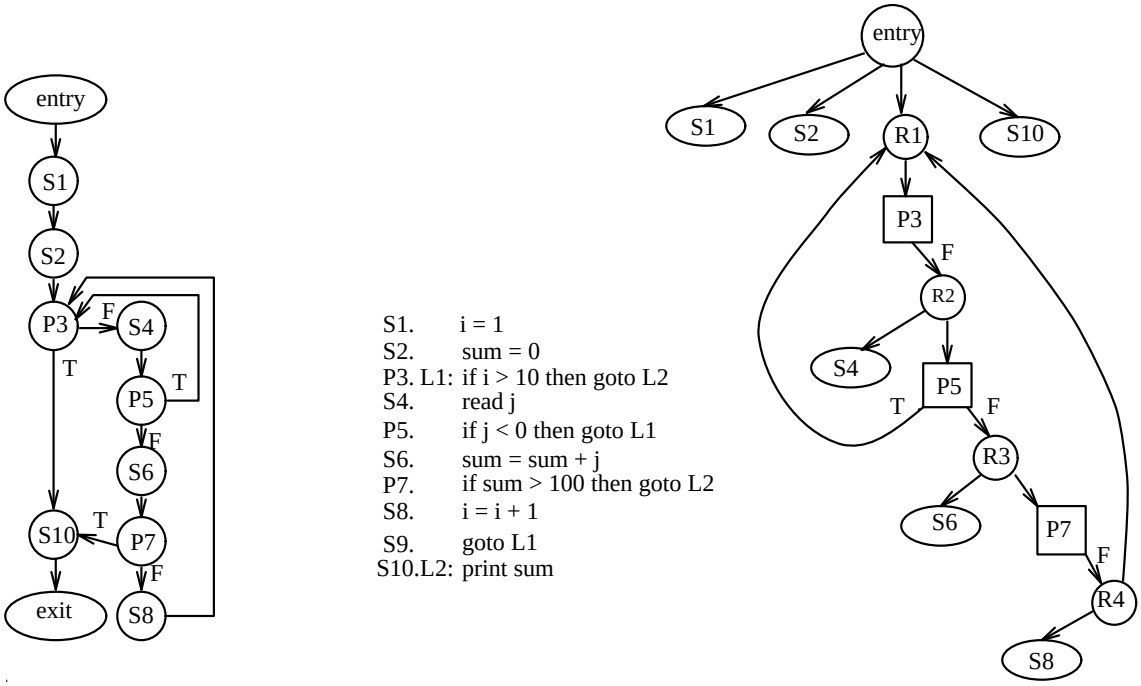

Figure 1: Program segment with its control flow graph (left) and its control dependence subgraph (right).

edges between nodes in the CDS to represent data dependencies. For example, in Figure 1, the DDS would contain an edge from S2 to S6 since S2 defines *sum* and S6 uses that definition of *sum*; likewise, there would be data dependence edges for each of the other definition-use pairs in the program. In Figure 1, we omit the DDS for the program.

3 Efficient PDG Construction

Our algorithm for constructing the PDG takes procedure P and produces the PDG for P in two steps. Step one constructs the CDS for P, and step two uses the CDS to construct the DDS for P. We address the two steps in the next two subsections.

3.1 Constructing the CDS

In this section, we overview our algorithm for constructing the CDS; details are given in [13]. Our algorithm `ConstructCDS`, sketched in Figure 2, accepts an abstract syntax tree (AST) for a procedure P and outputs the CDS for P. For the sake of presentation, we assume that P is written in a language containing the following types of statements: simple statements (*assignment*), structured statements (*if-else*, *while*), structured transfers of control (*continue*, *break*, *return*, and *exit*), and unstructured transfers (*goto*). Other constructs and statements are handled similarly. `ConstructCDS` uses a left to right preorder traversal of P's AST and takes appropriate actions as each node is encountered, and as each subtree in the AST is reduced. `ConstructCDS` handles two important tasks: (1) it creates CDS nodes that represent exact control dependencies in P, and (2) it encodes control flow for use in algorithms that require it.

`ConstructCDS` uses a stack, `CDStack`, and the usual set of stack operations, to maintain nesting, and therefore control dependencies. When `ConstructCDS` begins, an "entry" region that represents the entire procedure is created and placed on `CDStack`. Subsequently, nodes are added to the CDS as children of the node that is on the top of `CDStack` (the *active* node). Whenever a statement that begins a new region of control dependence, such as a structured statement or *label*, is found, a new node is added to the CDS and pushed onto `CDStack`. Subsequent statements, added under this new active node, are then properly nested. When the end of a structured statement is detected, `CDStack` is popped.

In most cases, our node order in the CDS encodes control flow implicitly. From a parent region, control flow moves to the leftmost child, then left to right among siblings until it reaches a region or predicate node. At predicate nodes control flows down outgoing control dependence edges. By ordering nodes as they are created, `ConstructCDS` preserves this implicit control flow.

```
algorithm   ConstructCDS
input       AST : abstract syntax tree for procedure P with root pgm
output      CDS : control dependence subgraph for P
declare     CDStack :  stack of information about region nodes
            Push(N), Pop(N) : push and pop operations on CDStack
            AdjustFlag:  boolean
            LabelTable :  table to record labels for control flow edges
            AddNode(N1, N2, L) : create CDS node N1, add it to CDS under N2 with label L
            (active:  the region on top of CDStack)

 begin
    while more AST nodes do
       Get ASTNode
       case ASTNode is
          pgm:
             create entry region as root of CDS; Push(entry)
             create exit region for later use
          assignment:
             AddNode(statement, Active, -)
          while:
             AddNode(header region, Active, -); Push(header region)
             AddNode(predicate, header region, -);
             AddNode(body region, predicate, true); Push(body region)
          end while:
             Pop(body region)
             resolve unresolved nodes
             Pop(header region)
             replace top of CDStack with while follow region if necessary
             set AdjustFlag if required
          if-else:
             AddNode(predicate, Active, -); Push(predicate)
          begin if-clause/else-clause:
             AddNode(region, Active, true or false); Push(region)
          end if-else:
             Pop(if-else region)
             replace top of CDStack with if-else follow region if necessary
          end if-clause/end else-clause
             list unresolved nodes
             Pop(if/else-clause regions)
             if else-clause region has no children remove it from CDS
          structured control transfer:
             AddNode(statement, Active, -)
             calculate follow information and update CDStack
             insert special flow and dependence edges
          goto:
             AddNode(statement, Active, -); update LabelTable; set AdjustFlag
          label:
             AddNode(label region, Active, -); Push(label region); update LabelTable
       end case;
    end while;
    add exit region to CDS under Active
    resolve remaining unresolved nodes and control dependence edges for breaks;
    if AdjustFlag then call Adjust
 end.
```

Figure 2: Algorithm to construct the CDS of a PDG, complete with control flow edges.

In other cases, control flow must be represented explicitly, by adding *control flow edges*. These cases fall into two categories. First, when the ends of structures are reached, and no sibling or child node is implicitly the next statement, a control flow edge must be inserted. `ConstructCDS` creates such edges, by detecting *unresolved nodes* and identifying their targets. Unresolved nodes are nodes whose successors have not yet been encountered in the AST traversal. Unresolved nodes are easily detected because they are the final nodes created before the ends of structures are encountered. When `ConstructCDS` detects such nodes, it places them on an *unresolved node list* associated with the active node. When a node is popped from `CDStack`, unresolved nodes listed with it are listed with its predecessor on the stack (the new active node). In this way lists are passed up the graph. Whenever `ConstructCDS` creates a node (other than an else-clause region node), if the active node has unresolved nodes listed with it, the algorithm creates edges between those unresolved nodes and the new node. Furthermore, whenever `ConstructCDS` reduces a *while* structure, it creates edges between unresolved nodes and the *while* region. Once edges have been created for unresolved nodes, the nodes are removed from the list they appeared on.

The second type of explicit control flow edge accounts for control flow associated with control transfer statements such as *continue* and *goto*. These edges are easily dealt with because their targets are explicit. We postpone discussion of these edges until the next sections.

`ConstructCDS` consists of a loop that repeatedly reads AST nodes until finished. A `case` statement embedded in the loop handles each type of AST node. For example, when the *pgm* node is encountered, `ConstructCDS` creates the *entry* region, pushes it onto `CDStack`, and creates an exit node for later use. Most cases make use of a procedure called `AddNode`, whose arguments are a new node N1, an existing node N2, and a label L. `AddNode` creates new node N1, and makes it a child of N2. If L is non-null, it denotes the label attached to the edge between N1 and N2. For example, `ConstructCDS` uses `AddNode` to add a statement node to the CDS under the active region when it encounters an *assignment* statement node. As a further example, when `ConstructCDS` encounters a *while* statement, it places a header region in the CDS under the active region, stacks the header region, inserts a predicate node under the header region, creates a "true" child region under the predicate, and stacks it. When the algorithm detects the end of a *while* statement, it pops regions from the stack and creates explicit control flow edges.

Figure 3 provides an example of a structured program, its AST, and the CDS `ConstructCDS` creates for that AST. When `ConstructCDS` sees the *pgm* node, it creates *entry* and *exit* regions, and pushes *entry* onto `CDStack`. Next, `ConstructCDS` creates statement nodes S1, S2, and S3 and makes them children of *entry*. The algorithm next translates the *while* statement into nodes R1, P4, and R2, and stacks R1 and R2. Subsequent statements, through the end of the *while* loop, are all control dependent upon the loop predicate, and thus nested under its "true" predicate region. `ConstructCDS` places the first *if-else* structure under R2, and the second under the "true" path from the first, reflecting control dependencies properly. When `ConstructCDS` encounters the end of the innermost *if-clause*, represented by region R4, it lists unresolved node S9 with R4. When the algorithm pops R4 from `CDStack`, it passes the list up to P8. Similarly `ConstructCDS` lists S10 with R5 and passes it up to P8. The algorithm continues to pass this list up as nodes are popped until the *while header* region R1 is encountered, at which point it creates edges (S9,R1) and (S10,R1). When `ConstructCDS` finds it has read the last AST node, it connects the exit node into the CDS and terminates.

3.1.1 Handling Procedures with Structured Transfers of Control

`ConstructCDS` encodes structured transfer statements as children of the active region, and determines their effects on control flow and control dependence. Where control flow is concerned, `ConstructCDS` creates explicit control flow edges from nodes representing *exit* and *return* statements to the exit region. `ConstructCDS` also inserts explicit control flow edges from nodes representing *continue* statements to their enclosing *while* header regions, found on `CDStack`. Finally, the algorithm associates nodes representing *break* statements with a list on the enclosing *while* header region. Later, `ConstructCDS` adds these nodes to the unresolved node list on the parent node of the header region.

Transfers have more complex effects upon control dependencies. Consider what occurs, for example, when a *continue* is enclosed in the *if-clause* of *if-else* statement S, such that the *if-clause* is executed when predicate P is true. When P is true, control flows immediately to the enclosing loop header. When P is false, however, statements after S are executed. Thus, statements after S are control dependent upon P. The idea of *follow* regions is presented by Ballance and Maccabe[3] to account for such control dependencies. A follow region summarizes the control dependencies of statements occurring after a compound statement. For example, the code after S constitutes a region that follows S and is control dependent on P false. The use of follow regions applies to nested statements as well.

`ConstructCDS` uses the concept of follow regions to construct the CDS in the presence of structured control transfer statements. When `ConstructCDS` encounters a *break* or *continue* statement that is enclosed in an *if-clause* (*else-clause*), `ConstructCDS` records the predicate path that leads to the *else-clause* (*if-clause*). The algorithm then attaches that predicate path to

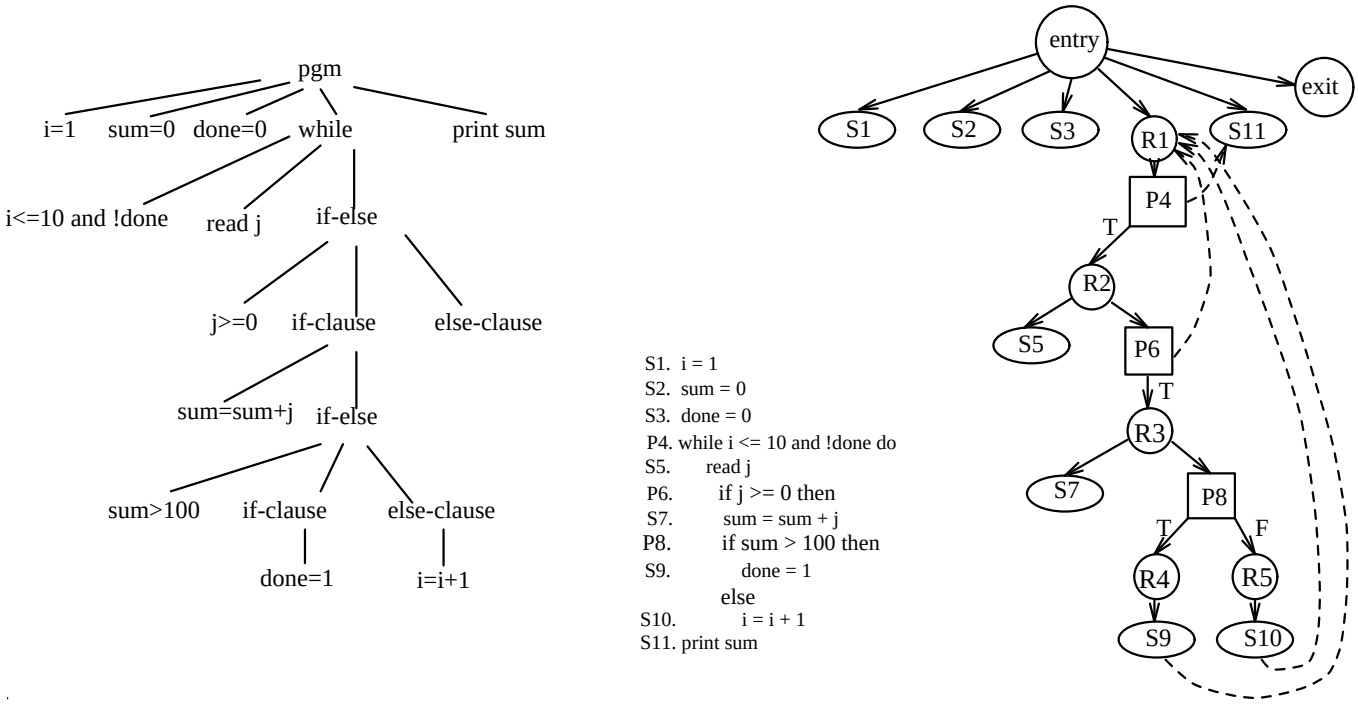

Figure 3: Structured program segment, its abstract syntax tree and its CDS. The dashed lines in the CDS represent explicit control flow edges that we compute during construction of the CDS; other control flow edges are implicit due to our ordering of nodes in the CDS.

each *if-else* region listed in `CDStack` within the enclosing *while* loop. Subsequently, when `ConstructCDS` encounters the end of an *if-else* statement it checks the top of `CDStack`. If the top entry has a follow, the top is popped and the next entry on the stack is replaced by the follow. If the top entry has no follow, the region now on top is the proper active region so no further action is necessary. `ConstructCDS` deals with *exit* and *return* statements similarly, but since these statements transfer control out of the procedure, they induce dependencies on code following enclosing *while* loops as well as enclosing *if-else* statements, so their follows are recorded with all enclosing structures.

Two additional control dependence effects must be considered. First, a *break* statement induces a dependence edge from its enclosing predicate to its enclosing loop header. `ConstructCDS` detects and adds this edge. Second, a conditionally executed *continue* statement does not by itself induce any control dependencies on an enclosing loop header region. However, when a loop contains both a *continue* statement and a transfer out of the loop, a dependence edge to the header region may be required. Our algorithm handles this case by detecting loops containing both *continues* and *breaks*, and setting a flag called `AdjustFlag`. When `AdjustFlag` is set, `ConstructCDS` calls procedure `Adjust` with a loop header region to have loop dependencies corrected, for each such loop detected.

Figure 4 displays a procedure, along with its AST and CDS, that contains structured control transfers. When `ConstructCDS` encounters the *continue* statement in this AST, it creates a *continue* statement node (S6) in the CDS under the active node, R3. The algorithm notes that the follow region for the *continue* is the node reached by edge P5-false, so it places P5-false in `CDStack` with the entry for P5. `ConstructCDS` then creates a control flow edge from S6 to R1. Later in the parse, when `ConstructCDS` pops P5 from `CDStack`, and notes the non-empty follow, it calculates the target of P5-False (R4), and replaces the top region on the stack (R2) by R4. Subsequent statements become descendants of (control dependent on) R4, as they should.

When `ConstructCDS` encounters the *break* statement (S9), it acts similarly. However, instead of inserting a control flow edge from S9 to the loop header (R1), `ConstructCDS` adds S9 to a list associated with R1. When the *while* statement is reduced, S9 is placed on the list of unresolved nodes associated with *entry*. This ensures that a control flow edge is created from S9 to S11 when S11 is created. `ConstructCDS` also creates the control dependence edge from R6 to R1.

Note that because the while loop contains both a *break* and a *continue* statement, `ConstructCDS` sets

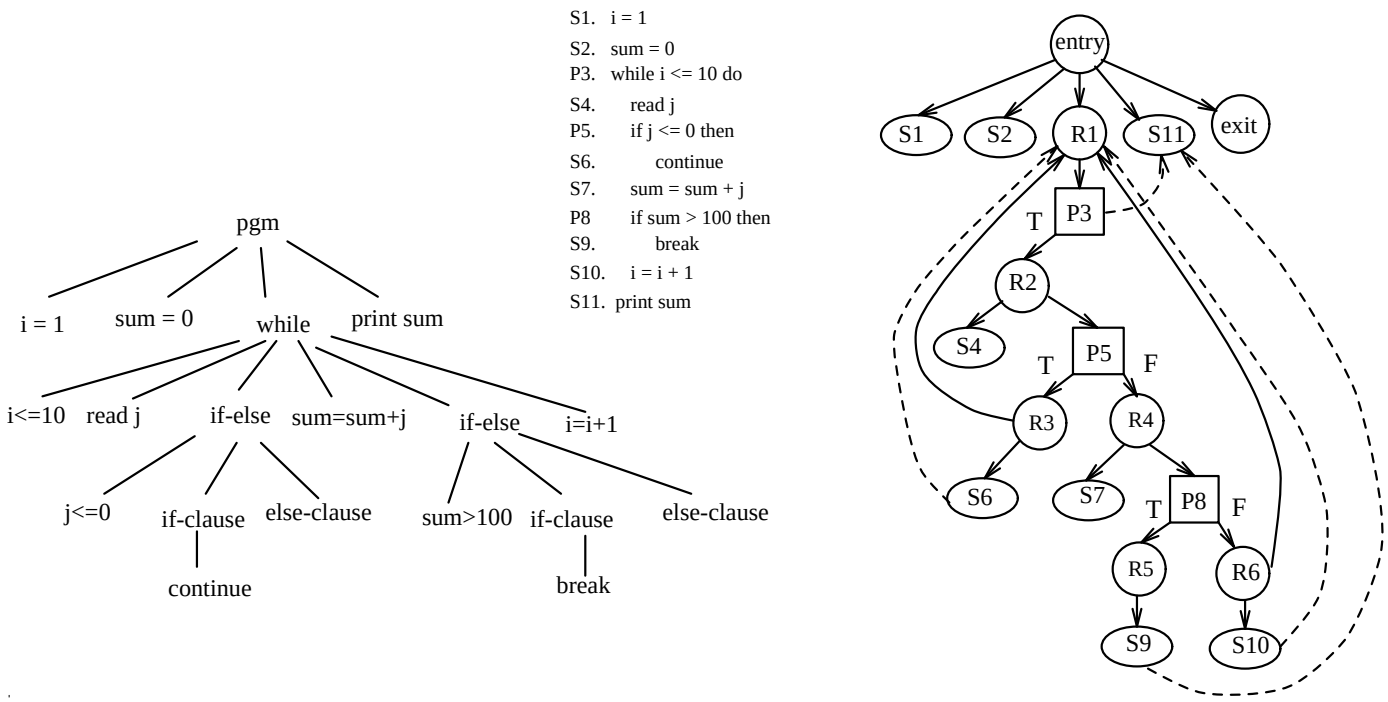

Figure 4: Program with structured transfers of control, its abstract syntax tree and its CDS. The dashed lines in the CDS represent explicit control flow edges that we compute during construction of the CDS; other control flow edges are implicit due to our ordering of nodes in the CDS.

`AdjustFlag` and calls `Adjust` with region R1. `Adjust` adds control dependence edge (R3,R1).

3.1.2 Handling Unstructured Procedures

When a procedure contains *goto* statements, we can no longer obtain its structure from the abstract syntax tree. In this case, `ConstructCDS` produces a graph in which control flow is represented precisely, but control dependence edges may need adjustment. It calls procedure `Adjust`, which computes exact control dependencies and adjusts the graph.

`ConstructCDS` uses a table, `LabelTable`, to record information about *gotos* and *label* statements. `ConstructCDS` inserts *goto* statements into the CDS as children of the active region. Also, on encountering a *goto*, `ConstructCDS` sets `AdjustFlag`, which causes `Adjust` to be called later. Since *label* statements may be targets of *goto* flow edges and dependence edges, `ConstructCDS` creates new regions for *label* statements, adds them to the CDS, and places them on `CDStack`. Subsequent nodes, up to the ends of enclosing structures, become children of these *label* statement regions. After all AST nodes have been processed, if *gotos* were present in the procedure, `ConstructCDS` uses `LabelTable` to insert control flow edges from *gotos* to their target label regions.

Consider, for example, what `ConstructCDS` does with the AST shown in Figure 5. When `ConstructCDS` encounters *label* statement L1, it creates label region R1 under the active region *entry*, pushes R1 onto `CDStack`, and makes a label table entry for L1, associating L1 with R1. When `ConstructCDS` encounters the *goto* in statement S4, it creates the S4 node, sets `AdjustFlag`, and makes a label table entry for L2. `ConstructCDS` treats subsequent *gotos* and *label* statements similarly. Upon reaching the end of the AST, `ConstructCDS` finds `AdjustFlag` set, and uses `LabelTable` to construct additional control flow edges. For example, the algorithm finds S4 in the `LabelTable` entry for *label* L2, and inserts edge (S4,R5). `ConstructCDS` then calls `Adjust` to render the control dependencies in the CDS exact.

3.1.3 Adjusting Control Dependencies

If `ConstructCDS` encounters unstructured *goto* statements during traversal of the AST, some regions may not be nested correctly with respect to control dependencies. In this case, the CDS produced by `ConstructCDS` is a *partial* CDS. Procedure `Adjust`, shown in Figure 6, is called to adjust control dependencies. `Adjust` uses the implicit and explicit control flow edges present in the partial CDS to compute

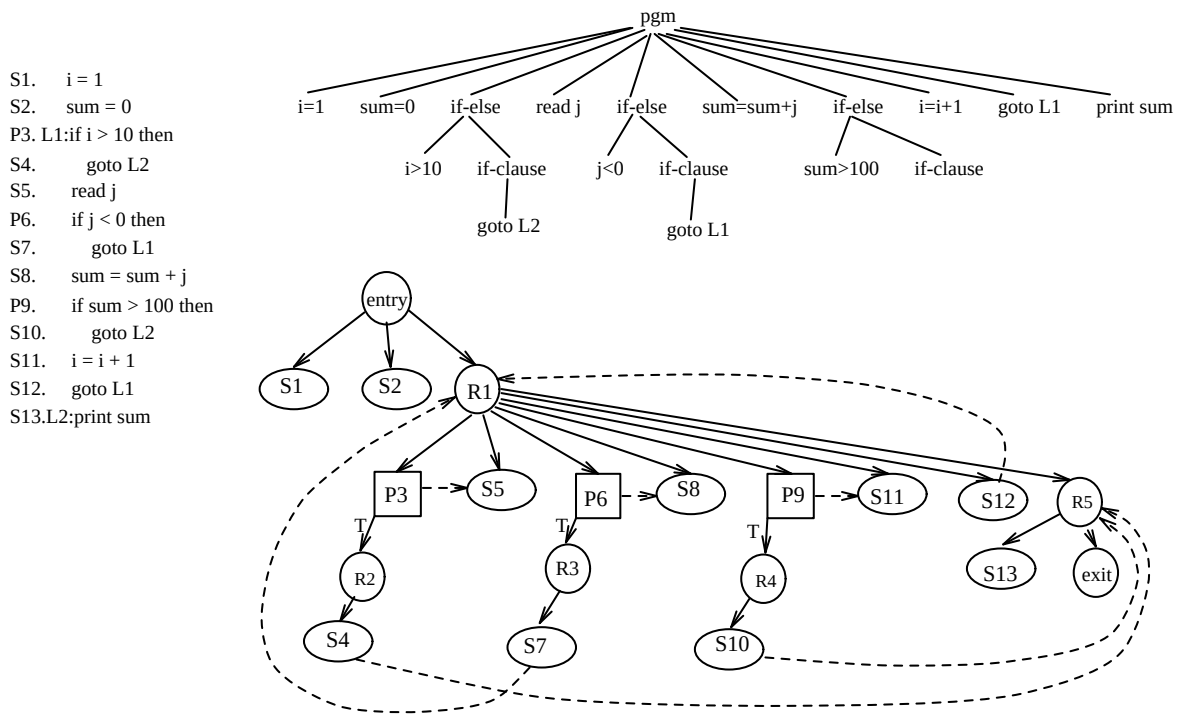

Figure 5: Program with goto statements, its abstract syntax tree and its partial CDS. Dashed lines in the CDS represent explicit control flow edges that we compute during construction of the CDS; other control flow edges are implicit due to our ordering of nodes in the CDS.

post-dominator information and exact control dependencies for each node. We use a technique developed by Cytron, Ferrante, Rosen and Wegman[6] to identify dominance frontiers, the control dependencies among the nodes in the partial CDS.

Procedure `Adjust` visits each node N in the partial CDS. As each node is visited, `Adjust` compares exact control dependencies with those present in the partial CDS. If they differ, N is removed from its current region(s), added to the correct regions, and marked.

`Adjust`'s final step is to merge nodes into the same regions and eliminate redundant regions. All similarly labeled children of a predicate node are combined by creating a new region that is control dependent on that predicate and label, and inserting the similarly labeled children into the new region. These children are ordered in the new region according to control flow. Redundant nodes are also eliminated in this final step.

Consider the actions of procedure `Adjust` when applied to the partial CDS shown in Figure 5. In the first step, `Adjust` uses the control flow information in the partial CDS to compute the control dependencies for each node in the partial CDS. `Adjust` then considers each node in the partial CDS and finds that S5, P6, S8, P9, S11 and S12 have computed control dependencies that differ from those expressed by the partial CDS. In particular, S5 and P6 are control dependent on P3-false and thus are removed from region R1, made control dependent on P3-false, and P3-false is marked. When `Adjust` considers marked nodes, all statements that are control dependent on P3-false are combined into a single region. Similar actions are taken for nodes S8, R9, S11 and S12. The resulting CDS is given in Figure 7.

3.2 Constructing the DDS

After the CDS is constructed using our technique described in section 3.1, we perform data flow analysis on it, and use the data flow sets to construct the DDS. The DDS may contain three types of data dependence edges, expressing flow-, anti- and output-dependence, depending on the application. *Flow-dependence* edges are added to the CDS from nodes containing definitions of a variable to nodes containing reachable uses of that variable. *Anti-dependence* edges connect nodes containing uses of a variable to definitions of that variable that follow. *Output-dependence* edges connect definitions of the same variable. We have developed both forward and backward data flow analysis algorithms that use the CDS. Here, we present our algo-

```
procedure   Adjust(R)
input       R : header region of CDS|_R to update
output      CDS with exact control dependencies

begin
   Compute CD(N) for each N in CDS|_R using CF
   foreach node N in CDS|_R do
      if CD(N) and control dependence of partial CDS
            differ then
         Remove N from current regions
         Add N to CD(N)
         Mark N as changed
      foreach marked N do
         Combine and eliminate redundant regions
end Adjust
```

Figure 6: Procedure to convert partial CDS to CDS with exact control dependencies.

rithm to compute reaching definitions, which is used to compute flow dependence information for the DDS; other data flow analysis algorithms are similar[13].

To compute sets of definitions that reach each statement in the program, we first compute local definition information, and then we propagate it throughout the program using the implicit and explicit control flow edges in our CDS until the sets stabilize. The number of iterations required depends on the loop nesting in the program. Our algorithm, `ReachingDefs`, is given in Figure 8. `ReachingDefs` assumes that local data flow information, consisting of the usual GEN and KILL sets, has been computed and is attached to CDS nodes. The GEN set consists of the definition, if any, in that statement (node); the KILL set consists of all other definitions of the GEN set's variable in the program. The GEN set is easily computed as the CDS is built. Using the GEN sets, KILL sets are computed for each statement node. Predicate nodes and region nodes have neither GEN nor KILL sets.

To propagate data flow information, we use IN and OUT sets where required. The IN set of a node consists of those definitions that reach the point immediately before the statement; the OUT set consists of those definitions that reach the point immediately after the statement. Since IN and OUT sets for a statement node may differ, we require both of these at each statement node in the CDS. However, since region and predicate nodes have identical IN and OUT sets, we use the OUT set to represent both of them.

On each iteration, algorithm `ReachingDefs` considers each node N in the CDS and computes its IN and OUT sets using either of two sets of equations depending on the nodes's type. If N is a region or predicate node, OUT[N] is computed as the union of the OUT sets of its control flow predecessors. If N is a statement node, IN[N] is the union of the OUT sets of its control flow predecessors and OUT[N] is synthesized using IN[N], GEN[N] and KILL[N].

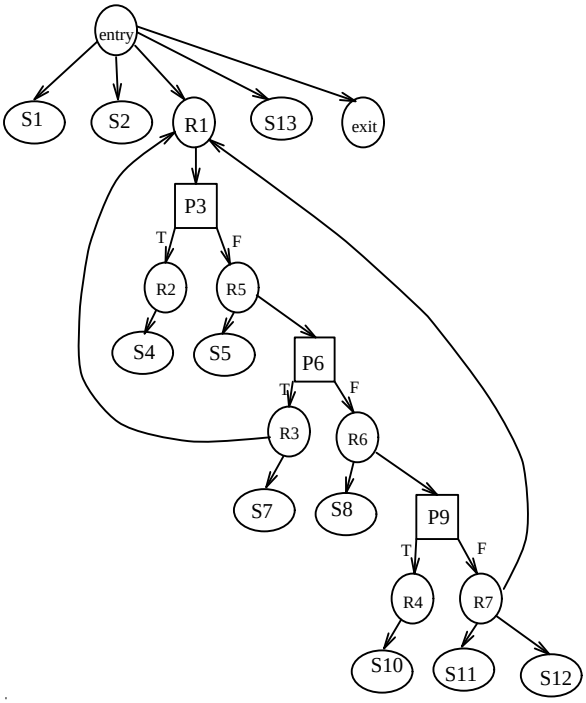

Figure 7: The CDS for the program in Figure 5 with exact control dependencies (control flow edges omitted).

After data flow analysis is completed, flow-, anti- and output-dependence edges are added to the CDS to get the DDS. Additionally, data flow sets are attached to each node in the graph. For simplicity, we omit the DDS from our figures.

4 Implementation

We have implemented our technique for constructing the PDG during parsing by incorporating it into the Free Software Foundation GNU C compiler version 2.3.3. Our implementation handles structured programs, along with structured and unstructured transfers of control. We are currently implementing the Ferrante, Ottenstein and Warren[8] method to facilitate experimental comparisons.

Our implementation handles programs with multiple procedures. For each procedure P, as the compiler constructs the abstract syntax tree for P, the implementation constructs the CDS for P. At the same time, the implementation collects local data flow information, and attaches GEN, KILL, DEF, and USE sets to CDS nodes. The implementation then performs data flow analysis to obtain the DDS for P. We have implemented both backward and forward data

```
procedure    ReachingDefs
input        control dependence subgraph (CDS),
             with control flow edges (CF),
             GEN/KILL sets computed for each
             statement node
output       reaching definitions at each node
declare      IN sets for each statement node
             OUT sets for each node

begin
   while changes do
      foreach node N in CDS do
         if N is a region/predicate node then
            OUT[N] = ∪ OUT[P], P is a control flow
               predecessor of N
         else
            IN[N] = ∪ OUT[P], P is a control flow
               predecessor of N
            OUT[N] = GEN[N] ∪ (IN[N] - KILL[N])
         endif
      endfor
   endwhile
end ReachingDefs
```

Figure 8: Algorithm to compute reaching definitions using the CDS and control flow edges.

flow algorithms, collecting live variable and reaching definition information, respectively. Our implementation works in an X-windows environment, and lets the user graphically view the PDGs created. We have used the PDGs created by our technique to facilitate data flow testing, to collect program trace information, and to construct slices of procedures. We also use the individual PDG's to construct a *unified interprocedural graph*[14] on which we perform interprocedural data flow analysis, program slicing and data flow testing. Our interprocedural analysis handles aliasing due to formal reference parameters and first level pointers.

5 Using Control Dependencies

There are many testing and analysis techniques that use control dependence information. Korel[16] uses the program dependence graph to identify test adequacy. Gupta and Soffa[9] use control dependence information to generate a reduced test set to satisfy test adequacy criteria. Their technique orders program statements and definition-use pairs that are then used to generate test cases. Duesterwald, Gupta and Soffa[7] use control dependencies to define a rigorous data flow testing criterion requiring that any satisfied definition-use pair must have had some influence on an output statment. They present three techniques, using static and dynamic slicing, for varying levels of precision and cost to determine test coverage. Harrold and Malloy[12] use the PDG for data flow testing using a technique to perform dynamic data flow analysis to satisfy adequacy criteria directly on the PDG.

Control dependence information has also been used for regression testing. Gupta, Harrold and Soffa[10] use both control dependencies and data dependencies to identify those definition-use pairs that must be retested after program modifications. Binkley[5] uses interprocedural control dependencies to identify a smaller version of the modified program that must be retested. Bates and Horwitz[4] define PDG-based adequacy criteria and propose new techniques based on slicing for regression testing after program modifications. They identify a subset of existing test cases that can be reused for testing. Schatz and Ryder[17] use the PDG to define a "condition-preserving" slice that is used to detect race conditions in parallel programs.

Many variations of program slicing, both static and dynamic, have been used for program analysis and testing. Although the original work on program slicing[19] used the control flow graph, later algorithms on the PDG[8] are more efficient since a PDG contains explicit control dependence information. Static program slicing has been extended to interprocedural slicing[15] using an extended form of the PDG that represents an entire program. Agrawal and Horgan[1] examine several approaches to dynamic slices and introduce the idea of a "reduced" dynamic dependence graph to handle the problem of size. Agrawal, DeMillo and Spafford[2] extend this work to handle dynamic slicing in the presence of unconstrained pointers.

6 Conclusion

We have presented an efficient algorithm that constructs a PDG in two phases: in the first phase it constructs a CDS from a procedure's abstract syntax tree, and in the second it uses data flow analysis on the CDS to obtain the DDS. Our algorithm handles programs with structured statements, and structured transfers of control, without requiring use of a control flow graph or post-dominator information. For programs containing unstructured transfers of control, an additional step adjusts the partial CDS constructed by the first phase of our algorithm, creating an exact CDS. By ordering nodes in the PDG, our technique encodes control flow for most statements explicitly. When necessary, we insert edges depicting control flow explicitly. Our approach has several advantages. Since in most cases we do not need a control flow graph or post-dominator information to construct our PDG, our technique is efficient. Moreover, because our PDG contains control flow information, it supports applications that would normally need to resort to control flow graphs, such as data flow analysis, test case generation, regression testing, and dynamic execution profiling. We have implemented our algo-

rithm by incorporating it into the GNU C compiler, and used PDGs created by it to collect program trace information, construct slices, and perform data flow testing. We are currently exploring applications to regression testing and testing object-oriented programs, via unified interprocedural graphs.

Acknowledgements

We wish to acknowledge Robert Ballance and Barney Maccabe for introducing the idea of a follow region for structured control transfers and the notion of "adjusting" the partial CDS in the presence of goto statements. We would also like to thank Rama Tummala and Amar Yalavarthy who implemented the `ConstructCDS` algorithm and improved its presentation. Finally, we thank the reviewers, who made many helpful suggestions that improved the paper.

References

[1] H. Agrawal and J. Horgan, "Dynamic program slicing," *Proceedings of ACM SIGPLAN '90 Symposium on Programming Language Design and Implementation,* pp. 246-256, June 1990.

[2] H. Agrawal, R. DeMillo and E. Spafford, "Dynamic slicing in the presence of unconstrained pointers," *Proceedings of the Symposium on Testing, Analysis and Verification,* pp.60-73, October 1991.

[3] R. Ballance and B. Maccabe, "Program dependence graphs for the rest of us," Technical Report, University of New Mexico, November 1992.

[4] S. Bates and S. Horwitz, "Incremental testing using program dependence graphs," *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages,* January 1993.

[5] D. Binkley, "Using semantic differencing to reduce the cost of regression testing," *Proceedings of the Conference on Software Maintenance '92,* pp. 41-50, November 1992.

[6] R. Cytron, J. Ferrante, B. Rosen and M. Wegman, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems,* vol. 13, no. 4, pp. 451-490, October 1991.

[7] E. Duesterwald, R. Gupta and M. L. Soffa, "Rigorous data flow testing through output influences," *Proceedings of the 2nd Irvine Software Symposium (ISS'92),* pp. 131-145, March 1992.

[8] J. Ferrante, K. J. Ottenstein and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems,* vol. 9, no. 3, pp. 319-349, July 1987.

[9] R. Gupta and M. L. Soffa, "Automatic generation of a compact test suite," *Proceedings of the Twelfth IFIP World Computer Congress,* September 1992.

[10] R. Gupta, M. J. Harrold and M. L. Soffa, "An approach to regression testing using slicing", *Proceedings of the Conference on Software Maintenance '92,*pp. 299-308, November 1992.

[11] M. J. Harrold and B. A. Malloy, "Performing data flow analysis on the PDG", Technical Report 92-108, Clemson University, March 1992.

[12] M. J. Harrold and B. A. Malloy, "Data flow testing of parallelized code," *Proceedings of the Conference on Software Maintenance '92,* pp. 272-281, November 1992.

[13] M. J. Harrold, B. A. Malloy and G. Rothermel, "Efficient construction of program dependence graphs," Technical Report 92-128 Clemson University, December 1992.

[14] M. J. Harrold and B. A. Malloy, "A unified interprocedural program representation for a maintenance environment," *IEEE Transactions on Software Engineering*, to appear.

[15] S. Horwitz, T. Reps and D. Binkley, "Interprocedural slicing using dependence graphs," *ACM Transactions on Programming Languages and Systems,* v. 12, no. 1, pp. 26-60, January 1990.

[16] B. Korel, "The program dependence graph in static program testing," *Information Processing Letters,* vol. 24, pp. 103-108, January 1987.

[17] E. Schatz and B. G. Ryder, "Directed tracing to detect race conditions," LCSR-TR-176, Laboratory for Computer Science Research, Rutgers University, February 1992.

[18] R. M. Stallman, "Using and porting GNU CC," Free Software Foundation, Inc., Cambridge MA, pp. 73-77, February 1990.

[19] M. Weiser, "Program slicing," *IEEE Transactions on Software Engineering,* vol. SE-10, no. 4, pp. 352–357, July 1884.