Syntax-Directed Construction of Program Dependence Graphs [1]

Mary Jean Harrold and Gregg Rothermel
Department of Computer and Information Science
The Ohio State University
Columbus, OH 43210-1277
{harrold,grother}@cis.ohio-state.edu

Abstract

We present an algorithm that constructs program dependence graphs as a program is parsed. For programs that contain only structured transfers of control, our algorithm does not require explicit control flow or postdominator information to compute exact control dependencies. For programs that contain explicit transfers of control, our algorithm can determine whether these transfers of control are used in a structured way, and if so, compute control dependencies without explicit control flow or postdominator information. When transfers of control are ill-behaved, our algorithm adjusts the control dependence information computed during the parse, to obtain exact control dependencies. For many programs, our algorithm provides savings in time and memory, because it does not require prior computation of control flow or postdominator information. However, our algorithm also calculates control flow information during the parse, and incorporates this information into the program dependence graphs that it constructs; the resulting graphs have a wider range of applicability than graphs that do not contain this information.

Keywords: control dependence, control flow, data flow, program dependence graphs, software testing.

1 Introduction

Many program testing and analysis techniques rely on control dependence or data dependence information. Some techniques use control or data dependence information to select test data and determine test set adequacy [31], to extend data flow testing techniques [12], or to generate reduced test sets for programs [15]. Several techniques for regression testing [3, 7, 8, 16, 37, 38] use control or data dependence information to determine the retesting required after changes are made to a program. Additionally, techniques that integrate different versions of programs [28] require control and data dependence information. Finally, both static and dynamic slicing techniques require control and data dependence information [1, 2, 40]. Many testing and analysis techniques also require control flow information. For example, data flow analysis [4], data flow testing [14, 33, 34, 36], test case generation [32], regression testing [17, 25, 35, 42], and dynamic execution profiling [5] require control flow information.

A *program dependence graph* (PDG) [13], in which nodes represent statements or regions of code, and edges represent control or data dependencies, encodes both control and data dependence information. Most existing techniques for PDG construction [10, 13] rely on explicit control flow and postdominance information to identify control dependencies and compute data dependence information. In most cases when programs contain *goto* statements, explicit control flow and postdominance information are prerequisites for identifying control dependencies. However, some programming languages, such as Modula-2, do not allow *goto* statements. Even languages that allow *goto* statements, such as C and Ada, promote structured design, which limits the use of such statements. In a case study in which we analyzed 3066 C functions from the

[1] This work was supported in part by a grant from Microsoft, Incorporated and Data General Corporation, and by NSF under Grant CCR-9109531 and NYI Award CCR-9357811 to Clemson University and Ohio State University. A preliminary presentation of this work appeared as "Efficient construction of program dependence graphs," in *ACM International Symposium on Software Testing and Analysis '93.*

GNU C compiler gcc[2] [41], UNIX[3] utilities, SPEC benchmarks,[4] and Mothra software testing system [11], we found that over 95 percent of the functions contain no *goto* statements. Our research shows that for these programs, we can construct PDGs without the use of explicit control flow or postdominance information. Moreover, the resulting PDGs represent control flow in addition to control and data dependence information.

In this paper, we present an algorithm for constructing PDGs that, in most cases, does not require prior calculation of control flow or postdominance information, yet succeeds in capturing control flow information along with data and control dependence information. Our algorithm constructs a partial PDG, consisting of control dependence information, during the parse phase of compilation. For structured programs and programs that contain structured transfers of control, such as *continue*, *break*, *return*, and *exit*, the algorithm computes control dependencies without relying on explicit control flow or postdominance information. When programs contain *goto* statements, the resulting graph may require additional processing to obtain exact control dependencies. Our partial PDGs encode control flow, in most cases implicitly by node order, and when necessary through inclusion of explicit edges. To add data dependence information to a partial PDG, we perform data flow analysis on the partial PDG using this control flow information. Using this data dependence information, we add data dependence edges to the partial PDG to obtain the complete PDG.

Because our algorithm provides syntax directed construction of PDGs without first computing control flow or postdominance information, it requires less time and space than traditional approaches for constructing PDGs. Ballance and Maccabe also presented a syntax directed technique for computing control dependence [6]. Unlike their technique, our algorithm computes control dependencies correctly for control structures that contain exit statements or multiple transfers of control. Also, when possible, our algorithm identifies *gotos* that are used as structured exits, and computes control dependencies for such *gotos* without relying on explicit control flow or postdominance information. Furthermore, we present our algorithm in detail sufficient to guide its implementation. Finally, our algorithm differs from both traditional techniques and Ballance and Maccabe's technique, in that it also computes control flow information, and attaches it to the control dependence graph, so that this information is available to applications that require it.

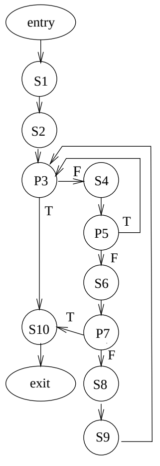
We have implemented a prototype tool that uses our PDG construction algorithm. Preliminary case studies using this prototype indicate that our approach to PDG construction can be effective in practice.

In the next section, we provide background information about PDGs. In Section 3, we present our technique for building PDGs by first discussing construction of control dependence subgraphs that also encode control flow, and then presenting the data flow analysis algorithms used to build data dependence subgraphs. We discuss our implementation and case studies in Section 4, and present our conclusions in Section 5. Appendix A gives a detailed version of our algorithm.

2 Background

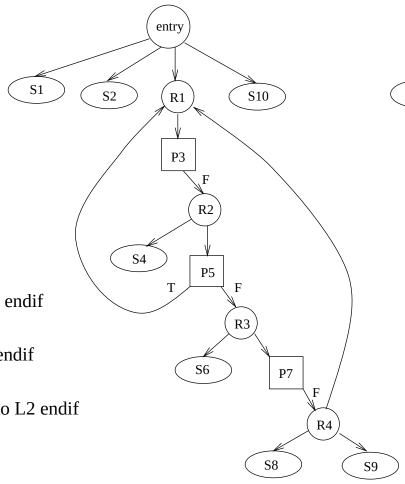
A program dependence graph[5] (PDG) encodes control dependencies in a *control dependence subgraph* (CDS). Control dependence is defined in terms of control flow graphs and the postdominance relation, as follows [13]. Given a control flow graph augmented with unique *entry* and *exit* nodes, and nodes W and V in that graph,

[2]Copyright (C) 1987, 1989 Free Software Foundation, Inc, 675 Mass Avenue, Cambridge, MA 02139.
[3]UNIX is a registered trademark licensed exclusively by Novell, Inc.
[4]Copyright (C) 1989, 1990, 1991, 1992 by Standard Performance Evaluation Corporation.
[5]Although called a *program* dependence graph, a PDG actually encodes information for a procedure.

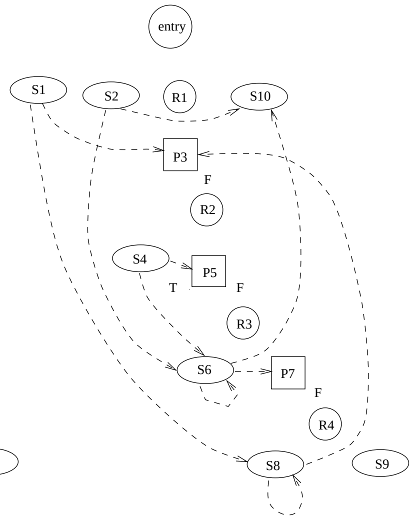


CFG

```
program Sums.1
begin Sums.1
 S1.          i = 1
 S2.          sum = 0
 P3.   L1:    if i > 10 then goto L2 endif
 S4.          read j
 P5.          if j < 0 then goto L1 endif
 S6.          sum = sum + j
 P7.          if sum > 100 then goto L2 endif
 S8.          i = i + 1
 S9.          goto L1
 S10.  L2:    print sum
end Sums.1
```



CDS



DDS

Figure 1: Program Sums.1, its control flow graph (CFG) on the left, its control dependence subgraph (CDS) in the center, and its data dependence subgraph (DDS) on the right. We show the CDS and DDS separately to facilitate presentation; however, the two subgraphs use the same set of nodes.

W is *postdominated* by V if every directed path from W to the exit (not including W or exit) contains V.[6] For statements (nodes) X and Y in a control flow graph, Y is *control dependent* on X if and only if (1) there exists a directed path P from X to Y with all Z in P (excluding X and Y) postdominated by Y and (2) X is not postdominated by Y. For example, Figure 1 depicts program `Sums.1` and its control flow graph. In the control flow graph, S10 postdominates all other nodes, P7 postdominates S6, and P3 postdominates S1, S2, S8, and S9. S8 is control dependent on P7-false, and S4 and P5 are control dependent on P3-false.

The nodes in a CDS represent single statements, or regions of code that have common control dependencies. The CDS for `Sums.1`, constructed using the Ferrante, Ottenstein, and Warren method [13], is shown in the center of Figure 1. A CDS contains several types of nodes. Statement nodes, shown as ellipses, represent simple statements. Predicate nodes, depicted as squares, correspond to statements from which two edges may originate. Node numbers labeling statement and predicate nodes correspond to statement numbers in the program. Predecessors of a node in the CDS are its *parents*, and successors of a node are its *children*. Children with the same parents are *siblings* of each other. Edges in the CDS represent control dependencies. Region nodes, shown as circles, group nodes that share the same control dependencies. For example, nodes S6 and P7 are each 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. The DDS for the program of Figure 1 is shown at the far right of the figure. A DDS is obtained by creating edges between nodes in the CDS to represent data dependencies. For example, in Figure 1, the DDS contains an edge from S2 to S6 because S2 defines `sum` and S6 uses `sum`; likewise, there are data dependence edges in the DDS for each of the other definition-use pairs in the program.[7]

[6] This definition of postdominance does not include the initial node on the path; thus, a node never postdominates itself.

[7] In the DDS of Figure 1, we show only *flow* dependencies; a flow dependence exists between a use of a variable and a definition that reaches it.

3 Constructing Program Dependence Graphs

Our algorithm for constructing PDGs takes a procedure P and produces its PDG in two steps. Step one constructs a CDS that contains control flow information for P, and step two uses the CDS to construct the DDS for P. We address these steps in Sections 3.1 and 3.2, respectively. For our presentation, we assume that P is written in a language that contains simple statements (*statement*), control structures (*if-else*, *while*), structured transfers of control (*continue*, *break*, *return*, and *exit*), and unstructured transfers of control (*goto*). We also assume that there is no statically unreachable code in the program.

3.1 Constructing Control Dependence Subgraphs

For structured procedures, control dependence relationships can be obtained directly from abstract syntax trees. If statement S is in the body of a *while* loop, S is control dependent with label *true* on the predicate that controls the *while* loop. If S is in the *then* part of an *if-else* statement, S is control dependent with label *true* on the predicate that controls the *if-else* statement. If S is in the *else* part of an *if-else* statement, S is control dependent with label *false* on the predicate that controls the *if-else* statement. When programs are structured, we can utilize this relationship between abstract syntax and control dependence to build control dependence subgraphs from the abstract syntax tree, without relying on control flow or postdominator information.

When procedures contain control transfer statements, the relationship between abstract syntax and control dependence is not initially obvious. However, in this case we can often still compute control dependencies from the abstract syntax tree. Occasionally, when programs contain ill-behaved control transfers, our computations will require adjustment. In any case, we can compute a program's control flow from the abstract syntax tree.

In this section, we present `ConstructCDS`, our algorithm for constructing a CDS. Figure 2 presents a high level view of the algorithm; Appendix A gives a more detailed view. We begin by discussing some general issues relevant to `ConstructCDS`, and factors relevant to representing control flow in control dependence subgraphs. We then discuss the algorithm in detail. To simplify our discussion, we present our algorithm as if it were traversing the AST. However, when implementing `ConstructCDS`, it is not necessary to create the entire AST before invoking the algorithm. Rather, the parser may serve as a driver for `ConstructCDS`, calling it with each AST node as the node is recognized.

`ConstructCDS` accepts an abstract syntax tree (`AST`) for a procedure P and outputs a CDS (`CDS`) for P that contains control flow information. `ConstructCDS` uses a left to right preorder traversal of the AST for P, and takes specific actions as it encounters each AST node. The algorithm handles two important tasks: (1) it creates CDS nodes and edges to represent control dependencies in P and (2) it encodes P's control flow for use by algorithms that require it.

`ConstructCDS` often requires information about the type of statement or region a node represents. Thus, we distinguish several types of nodes in a CDS. We classify statement nodes as 'statement', 'predicate', or 'exit', and we classify region nodes as 'entry', 'if-clause', 'else-clause', 'while header', 'while body', 'while exit', 'summary', or 'label'.

`ConstructCDS` uses a stack, `CDStack`, and the usual `Push()` and `Pop()` stack operations, to store information about region and predicate nodes; `CDStack` contains information on regions of code that `ConstructCS`

```
algorithm   ConstructCDS
input       AST : abstract syntax tree for procedure P with root pgm
output      CDS : control dependence subgraph for P, with control flow information
declare     ASTNode : ASTNodeType
            CDStack : stack of region/predicate nodes
            RegionTable : table of region nodes and associated predicate paths
            TransferList : table of predicate nodes and predicate paths for associated transfer statements
            Active : the region on top of CDStack
            LoopStack : stack of information about enclosing loops
            LabelTable : table to record labels for control flow edges
            AdjustFlag : integer
            GetASTNode() : returns the next node in the AST

begin
    while ( (ASTNode := GetASTNode()) and ASTNode.type ≠ end pgm ) do
        case ASTNode.type is
            pgm:
                create 'entry' node as root of CDS; push it onto CDStack
                create 'exit' node for later use
            statement:
                create 'statement' node as a child of Active region
            if-else:
                create 'predicate' node as a child of Active region; push it onto CDStack
            if-clause/else-clause:
                create 'if-clause'/'else-clause' node as a child of Active region with 'T'/'F' edge label; push it onto CDStack
            end if-clause/end else-clause
                pass unresolved node information up
                pop CDStack to discard label regions and the 'if-clause'/'else-clause' region node
            end if-else:
                pop CDStack to discard 'predicate' node;
                replace top node on CDStack with if follow region if necessary
                update predicate path information on TransferList for the topmost predicate on CDStack
            while:
                create 'while header' node as a child of Active region; push it onto CDStack and LoopStack
                create 'predicate' node as a child of Active region; push it onto CDStack
                create 'while body' node as a cd successor of 'predicate' node with 'T' edge label; push it onto CDStack
            end while:
                pop CDStack to discard 'label' nodes and 'while body' region node
                resolve unresolved nodes to 'loop header'
                pop CDStack to discard 'while predicate' node
                create 'while exit' node as a control dependence successor of 'while predicate' node with 'F' label
                resolve transfer-induced control flow and control dependence edges to loop header if necessary
                pop CDStack to discard 'while header' region
                replace top node on CDStack with while follow region if necessary
                update predicate path information on TransferList for the topmost predicate on CDStack
            continue/break/return/exit:
                create 'statement' node as a child of Active region
                update predicate path information on TransferList for the topmost predicate on CDStack
                insert control flow and control dependence edges if necessary
            goto:
                create 'statement' node as a child of Active region; update LabelTable
                handle as continue, break if possible; set AdjustFlag if necessary
            label:
                create 'label' node as a child of Active region; push it onto CDStack
        endcase
    endwhile (* end pgm encountered, loop exited *)
    discard 'label' regions on CDStack; add 'exit' node as a child of 'entry' region;
    resolve unresolved nodes
    if AdjustFlag then Adjust endif
    remove spurious regions
    return(CDS)
end ConstructCDS.
```

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

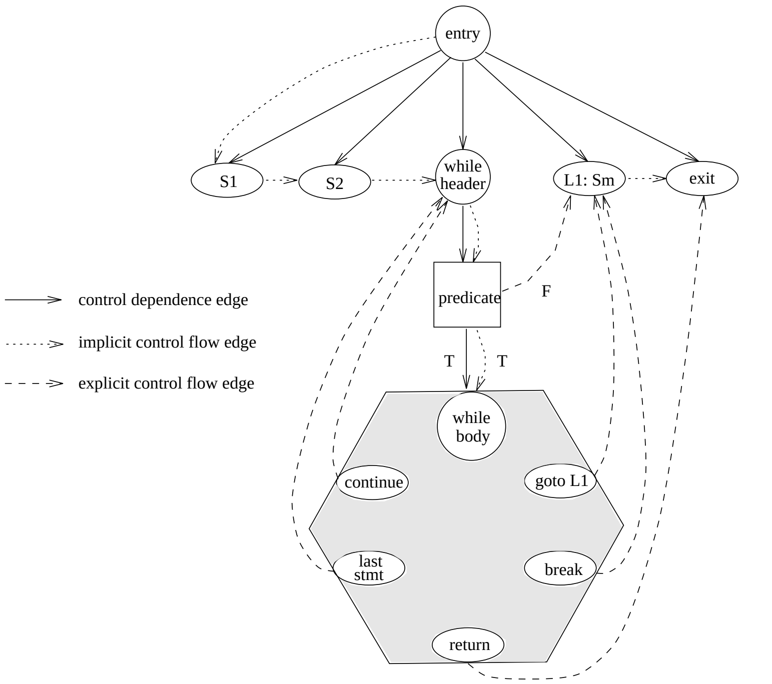


Figure 3: Control dependence subgraph with implicit and explicit control flow edges.

has not finished processing. We refer to the region at the top of `CDStack` as the active region, `Active`. `ConstructCDS` begins by creating an 'entry' region node and placing it on `CDStack`. Whenever the algorithm encounters a statement that begins a new region of control dependence, such as an *if-else*, *while*, or *label*, it creates a new region node, adds it to the CDS, and pushes the region node onto `CDStack`. Subsequently, `ConstructCDS` adds nodes to the CDS as children of `Active`. When the algorithm detects the end of a control structure, it pops `CDStack`.

In most cases, we represent control flow implicitly in the CDS by node order. From a region node, control flows to the leftmost child of that region, then from left to right among siblings until it reaches another region node or a predicate node. At predicate nodes, control flows along outgoing control dependence edges. By ordering nodes as it creates them, `ConstructCDS` preserves this implicit control flow. Consider the CDS of Figure 3. The shaded area represents the *while* loop's body, and contains several transfer of control statements; solid lines represent control dependence edges. Implicit control flow edges, shown as dotted lines, can be either labeled or unlabeled. Edge ('entry', S1) is an example of an unlabeled implicit control flow edge; edge ('predicate', 'while body' 'T') is an example of a labeled implicit control flow edge.

In other cases, we represent control flow explicitly by adding *control flow edges*. These cases fall into two categories. The first type of explicit control flow edge occurs at the ends of control structures, where no sibling node or child node is implicitly the next statement. `ConstructCDS` creates such edges by detecting unresolved nodes and identifying their targets. *Unresolved nodes* are nodes whose control flow successors have not yet been encountered in the AST traversal; they are easy to detect because they are the final nodes processed before the ends of control structures are encountered. When `ConstructCDS` recognizes that a node is unresolved, it places it on a list, `.cfunreslist`, associated with the `Active` node. When the algorithm pops a node `N` from `CDStack`, it adds unresolved nodes on `N.cfunreslist` to the `.cfunreslist` associated with `N`'s predecessor on `CDStack` (the new `Active` node). When `ConstructCDS` creates the node that represents the control flow successor of unresolved nodes in a particular control structure, it locates these unresolved nodes on `Active.cfunreslist`, and creates control flow edges from the unresolved nodes to the control flow successor. Similarly, when the algorithm encounters the end of a *while* statement, it creates edges between nodes on `Active.cfunreslist` and the 'while header' node to represent control flow backedges from the end of the *while* loop to the beginning of the loop. After `ConstructCDS` creates edges for unresolved nodes, it removes the nodes from the `.cfunreslist` on which they appeared.

Figure 3 uses dashed lines to depict explicit control flow edges. The *while* body contains a last statement in the loop (last stmt) that is not an explicit transfer of control. When `ConstructCDS` encounters the end of the *while* statement, it adds an unlabeled explicit control flow edge, (last stmt, while header), to the CDS, to represent the backedge from the end of the *while* loop to the beginning of the loop. Another explicit control flow edge that `ConstructCDS` adds at this point is the labeled edge (predicate, L1: Sm), which represents the flow of control when the *while* statement's predicate is false and the loop is exited.

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 never labeled. `ConstructCDS` handles these control flow edges easily because their targets are explicit. We postpone discussion of these edges until Sections 3.1.2 and 3.1.3. However, Figure 3 shows the unlabeled explicit control flow edges (continue, while header), (goto L1, L1: Sm), (break, L1: Sm), and (return, exit).

In the following section, we discuss the application of `ConstructCDS` to structured programs. In subsequent sections, we consider the actions required to handle structured and unstructured transfers of control.

3.1.1 Constructing CDGs for Structured Procedures

`ConstructCDS` uses several procedures to accomplish its tasks; Appendix A lists these procedures. Procedure `AddNode()` has three arguments: `NodeType T`, `CDSNodeType N`, and `EdgeLabel L`. `AddNode()` creates a new CDS node of type `T`, makes it a child of `N`, and returns the node. If `L` is not empty, it is the 'T' or 'F' label on the control dependence edge between `N` and the new node. Procedure `AddNode()` uses procedure `MakeCDSNode()` to creates a new CDS node. In addition to creating the new CDS node, `MakeCDSNode()` handles unresolved nodes: if there are unresolved nodes listed on `Active.cfunreslist` that require explicit control flow edges, `MakeCDSNode()` creates these control flow edges to the new node. After `MakeCDSNode()` returns, `AddNode()` calls `AddtoCDS()` to connect the new node into the CDS by creating control dependence edges. Finally, if the new node is a region node `AddNode()` adds it to `RegionTable`. `RegionTable` is used to compute control dependence conditions for statements that follow control statements that contain explicit transfers of control; we defer further discussion of the table to Section 3.1.2. Routines `Push()` and `Pop()` manage `CDStack`; `Pop()` also ensures that unresolved nodes listed with `Active` are listed with the next node down in `CDStack` when `Active` is popped. Function `GetASTNode()` returns the next AST node, `ASTNode`, found in a preorder traversal of the AST. Another stack, `LoopStack`, stores information about enclosing loops, and is manipulated by `PushLS()` and `PopLS()` routines. `Search()` accesses `LabelTable`, which keeps track of *goto* and *label* information. Finally, `ComputeFollow()` computes control dependence conditions for statements that follow control statements that contain explicit transfers of control. We defer further discussion of this routine to Section 3.1.2.

The main `ConstructCDS` routine consists primarily of a loop that reads AST nodes until it encounters the *end pgm* node. A `case` statement nested in the loop handles each type of AST node. We next discuss the actions that `ConstructCDS` takes for each type of AST node.

Program entry

When `ConstructCDS` encounters the *pgm* AST node, it creates the 'entry' node, pushes it onto `CDStack`, and creates the 'exit' node for use as the target of control flow edges from *return* and *exit* nodes.

Simple statements

When `ConstructCDS` encounters a *statement* in the AST, it uses `AddNode()` to create a 'statement' node and connect it into the CDS.

If-else statements

When `ConstructCDS` encounters an *if-else* statement, it first reads the *condition* node in the AST. Then, the algorithm inserts a 'predicate' node as a child of `Active`, and pushes the 'predicate' node onto `CDStack`. When the algorithm finds an *if-clause*, it inserts an 'if-clause' region node as a child of `Active` (the 'predicate' node) with label 'T', and pushes it onto `CDStack`; subsequent statements in the true part of the *if-else* statement will be added to this 'if-clause' region. On reaching *end if-clause*, `ConstructCDS` adds the node most recently created to `Active.cfunreslist` if that node is not a transfer node, and pops label regions from `CDStack`. The algorithm then pops `Active` ('if-clause') from `CDStack`, causing its `.cfunreslist` to be passed back to the *if-else* 'predicate' node on `CDStack`. The algorithm acts similarly when it finds *else-clause* and *end else-clause* AST nodes. Finally, when `ConstructCDS` encounters *end if-else*, it pops the *if-else* 'predicate' node. If the structure contained transfers of control, additional work is required; we defer discussion of this work to Section 3.1.2.

Figure 4 shows a snapshot of the CDS construction after an *if-else* statement is processed; the figure also gives the partial AST. Control flow edges from statements at the end of each branch of the *if-else* statement cannot be determined at this time, so `ConstructCDS` stores information about these unresolved nodes with `Active` on `CDStack` for later resolution. Shaded ellipses represent the unresolved nodes.

While statements

A *while* statement begins a new region of control dependence. Thus, when `ConstructCDS` encounters a *while* statement, it creates a 'while header' region node, adds it to the CDS as a child of `Active`, and pushes it onto both `CDStack` and `LoopStack`. `ConstructCDS` then reads the AST condition node, creates a 'predicate' node, with the condition just read, as a child of `Active`, and pushes the predicate node onto `CDStack`. The body of the *while* loop is a new region of control dependence. Thus, the algorithm creates a 'while body' region node, adds it to the CDS as a control dependence successor of the true branch of the 'predicate' node just created, and pushes this 'while body' region node onto `CDStack`. Subsequent statements encountered in the *while* loop will be added to the CDS as descendants of this 'while body' region. The snapshot on the left in Figure 5 shows a partial CDS at this point in the processing of a *while* loop; the figure also gives the partial AST.

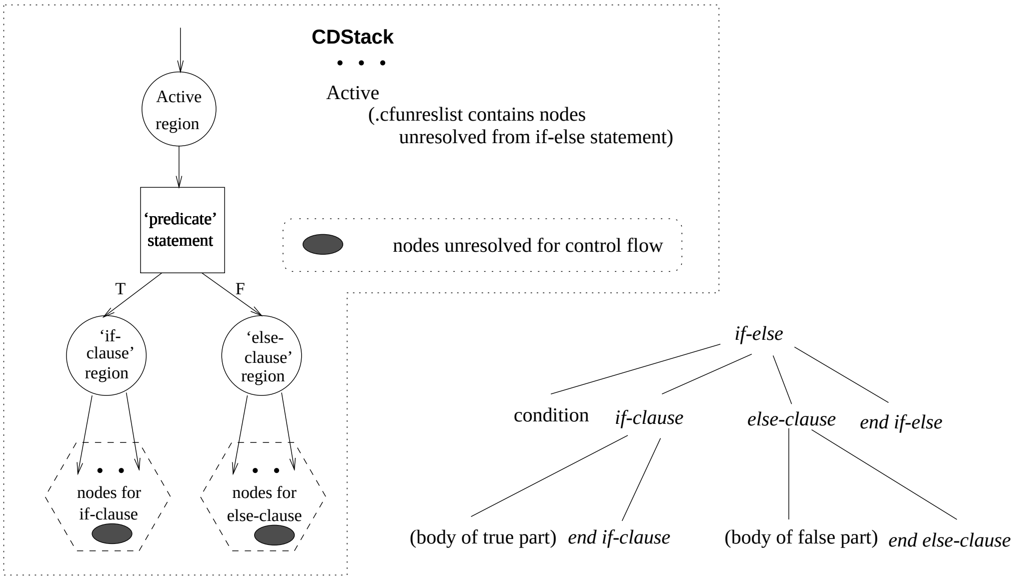


Figure 4: Snapshot of `CDS` and `CDStack` after an *if-else* statement is processed; the program is structured with its partial AST given on the lower right.

When the loop processing is complete, subsequent statements encountered will not be control dependent on the 'while body' region, or on any regions created for 'labels' in the loop. Thus, when `ConstructCDS` encounters an *end while* node in the AST, it discards any 'label' nodes it finds on top of `CDStack`, and then pops the 'while body' region node. There may be a single loop-ending unresolved node that was the last node created; if that node was not a control transfer statement, `ConstructCDS` adds it to `Active.cfunreslist`. At this point, all nodes that have implicit control flow to the beginning of the loop have been recorded on `Active.cfunreslist`. The algorithm now creates control flow edges from all nodes on `Active.cfunreslist` to the 'while header' node; it then reinitializes `Active.cfunreslist` to empty. The algorithm next pops the 'while predicate' node, creates a 'while exit' node in the CDS as a control dependence successor of the false branch of this node, and adds the 'while exit' node to `Active.cfunreslist` so that the control flow edge out of the loop can later be connected to the statement that follow the *while* loop.

After handling the 'while exit' node, `ConstructCDS` performs actions to account for the effects of control transfer statements, which we discuss in Sections 3.1.2 and 3.1.3. Next, `ConstructCDS` pops the 'while header' node from `CDStack`, and then calls `PopLS`, which pops the 'while header' node from `LoopStack`. If there are no transfers of control out of the loop, except for the normal end of the loop, then the statements in the body of the *while* loop are control dependent on the *while* 'predicate', and a control dependence edge connecting the 'while body' region node to the 'while header' region node must be added to the CDS. `PopLS()` detects whether this edge is necessary and if so, adds it to the CDS. Additional actions following the `PopLS` call are also associated with control transfers; we defer discussion of these actions to Section 3.1.2.

The snapshot on the right in Figure 5 shows `CDStack`, `LoopStack`, and a partial CDS after a *while* statement is processed. The control dependence backedge, ('while body' region, 'while header' region), is present if there are no statements, such as *break* or *exit*, that transfer control out of the loop. There is a control flow backedge (resolved, 'while header' region) for each loop-ending statement that is not an explicit

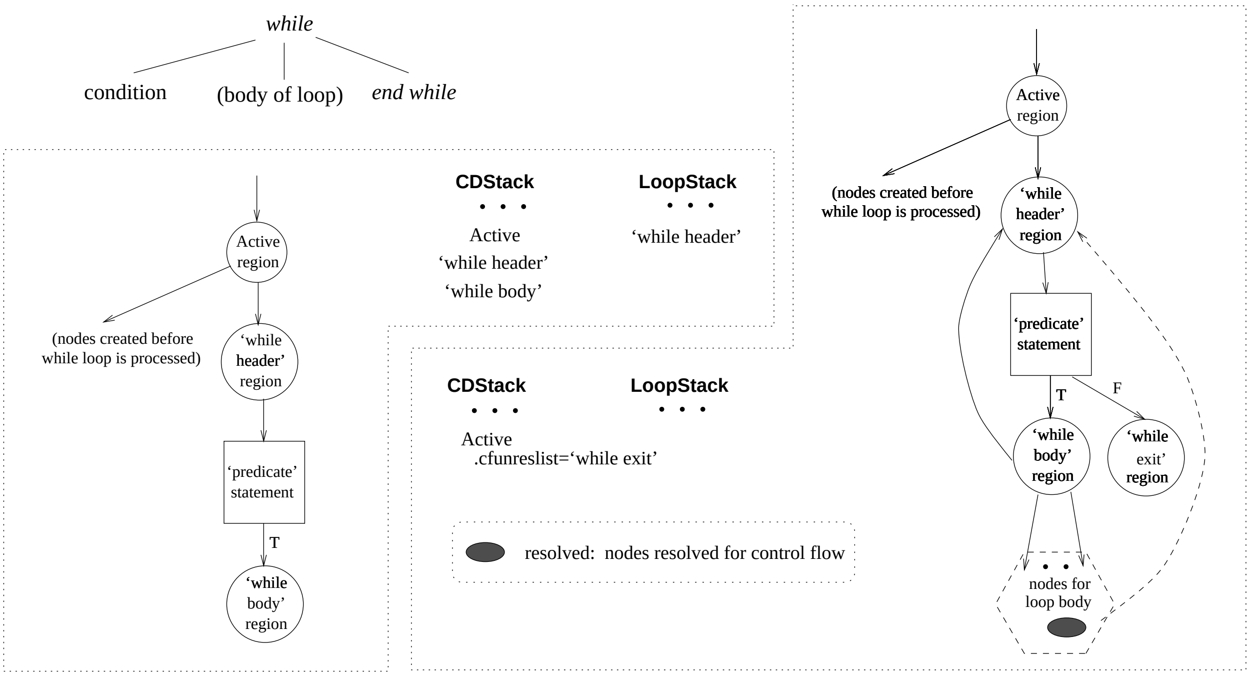


Figure 5: Snapshot of CDS and stacks during *while* statement processing (in the lower left) and after *while* statement processing (on the right); assumes that the program is structured with its partial AST given in the upper left.

---

transfer of control; such statements were stored on `Active.cfunreslist` during loop processing. Because we assume in this section that the program is structured, `Active.cfunreslist` contains only the 'while exit' region node, which will be resolved when the next statement is encountered.

Program exit

When `ConstructCDS` detects the *end pgm* AST node, it discards all 'label' regions on `CDStack`. The algorithm then attaches the 'exit' node to the CDS, and adds control flow edges from any remaining unresolved nodes to that 'exit' node. Sections 3.1.2 and 3.1.3 discuss additional work, necessitated by the presence of control transfers. Finally, `ConstructCDS` removes two types of spurious region nodes from the CDS: (1) region nodes that have no control dependence children, and (2) region nodes that have only a single region node as a control dependence successor. In each case, the algorithm adjusts control dependence and control flow edges that involve deleted region nodes. Note that the control dependence subgraph construction algorithm described in Reference [13] also performs a post-pass of the CDS to remove region nodes of the second type.

Example

Figure 6 gives a structured program, `Sums.2`, its AST, and the CDS that `ConstructCDS` creates for that AST. We now trace the operations of `ConstructCDS` for this example, omitting reference to actions that involve `TransferList` and `RegionTable` because they are not required when processing structured programs. When `ConstructCDS` sees the *pgm* node, it creates 'entry' and 'exit' nodes, and pushes the 'entry' node onto `CDStack`. Next, `ConstructCDS` creates 'statement' nodes S1, S2, and S3, and makes them children

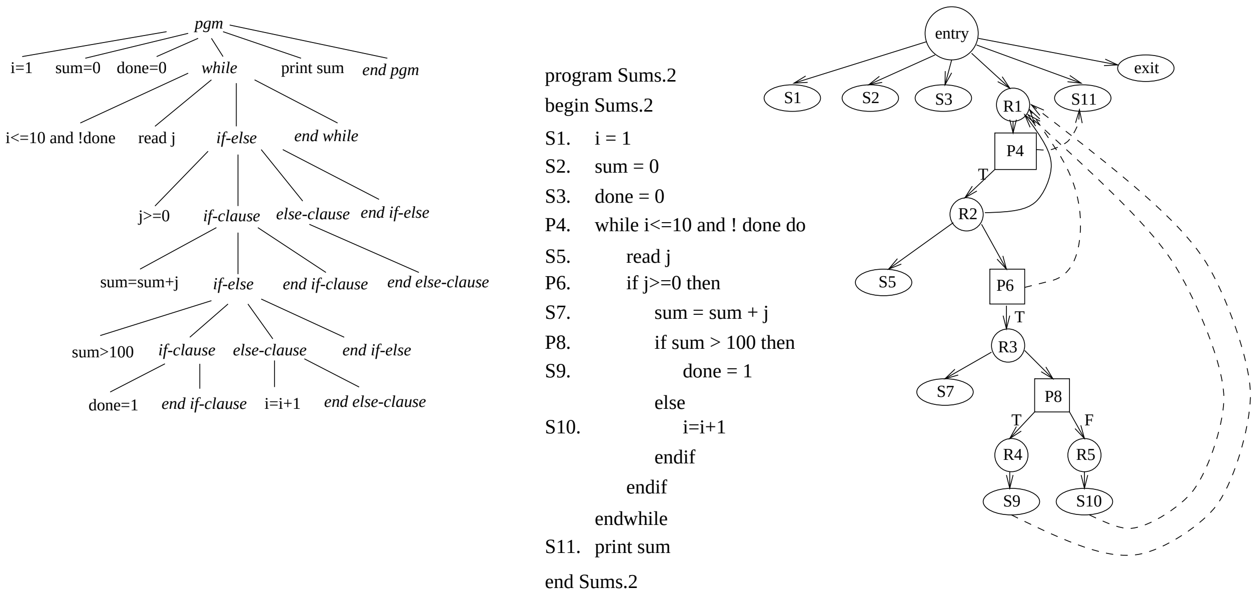


Figure 6: Structured program `Sums.2`, its AST, and its CDS. The dashed lines in the CDS represent explicit control flow edges that our algorithm computes during construction of the CDS; other control flow edges are implicit due to ordering of nodes in the CDS.

---

of the 'entry' node. When `ConstructCDS` encounters the *while* statement in `AST`, it creates 'while header' region node R1, adds it to `CDS`, and pushes it onto `CDStack` and `LoopStack`. The algorithm then reads the condition node in `AST`, creates 'predicate' node P4, and adds P4 to `CDS` and pushes it onto `CDStack`. Next, `ConstructCDS` creates 'while body' region node R2, connects it to P4 with label 'T' and pushes it onto `CDStack`. `ConstructCDS` next creates S5 and 'predicate' node P6 as children of R2; the algorithm also pushes P6 onto `CDStack`. When `ConstructCDS` encounters the *if-clause* AST node, it adds 'if-clause' region node R3 as a control dependence successor of P6 with label 'T', and pushes R3 onto `CDStack`. The next statements, similarly, yield 'statement' node S7, 'predicate' node P8, 'if-clause' region node R4 with label 'T', and 'statement' node S9; during the construction of this part of `CDS`, `ConstructCDS` pushes P8 and R4 onto `CDStack` in that order. At this point in the construction of `CDS`, `CDStack` contains 'entry', R1, P4, R2, P6, R3, P8 and R4.

When `ConstructCDS` encounters the end of the innermost *if-clause*, it lists the most recent node encountered, S9, on the `.cfunreslist` of R4, which is the current `Active` node. `ConstructCDS` next pops R4 from `CDStack`; the `Pop()` routine passes S9 down to `P8.cfunreslist`. `ConstructCDS` next encounters the *else-clause*, creates 'else-clause' region node R5 with label 'F' as a control dependence successor of P8, and pushes R5 onto `CDStack`. The algorithm then creates 'statement' node S10 as a child of R5. Detecting an *end else-clause* in `AST`, `ConstructCDS` adds S10 to the `.cfunreslist` of `Active`, R5, and pops R5; this passes S10 down to `P8.cfunreslist`. Both S9 and S10 are now on `P8.cfunreslist`. At this point, the algorithm encounters *end if-else*, pops P8 from `CDStack`, and passes `P8.cfunreslist` down to `R3.cfunreslist`. `ConstructCDS` then encounters an *end if-clause* node in `AST`, pops R3 from `CDStack`, and passes `R3.cfunreslist` down to `P6.cfunreslist`.

After processing the 'end if-clause' associated with statement P6, the algorithm detects an *else-clause*,

```
S1.   while  P1
S2.        if P2 then
S3.             continue
           endif
S4.        <statement>
      endwhile
```

Fragment A

```
S1.    while P1
S2.        if P2
S3.             if P3 then
S4.                  continue
                endif
S5.             <statement>
            endif
S6.         <statement>
       endwhile
```

Fragment B

```
S1.    while P1
S2.        if P2
S3.             if P3 then
S4.                  <statement>
                endif
S5.             exit
            endif
S6.         <statement>
       endwhile
S7.    <statement>
```

Fragment C

```
S1.    while P1
S2.        if P2
S3.             if P3 then
S4.                  break
                endif
S5.             exit
            endif
S6.         <statement>
       endwhile
S7.    <statement>
```

Fragment D

Figure 7: Code fragments that illustrate the need for *follow* regions.

creates 'else-clause' region node R6 as a control dependence successor of P6 with label 'F', and pushes it onto `CDStack`; R6 does not appear in Figure 6 because subsequent processing finds that it has no control dependence children, and removes it from `CDS`. `ConstructCDS` then encounters *end else-clause*, and adds the most recent node processed, R6, to `Active.cfunreslist`. The algorithm pops R6 from `CDStack`, and passes `R6.cfunreslist` down to `P6.cfunreslist`. At this point, `ConstructCDS` detects *end if-else*, pops P6 from `CDStack`, and passes `P6.cfunreslist` down to `R2.cfunreslist`.

Next, `ConstructCDS` encounters *end while* in the `AST`, pops 'while body' region node R2, and passes `R2.cfunreslist` down to `P4.cfunreslist`. The algorithm then uses `P4.cfunreslist` to create control flow edges (S9,R1), (S10,R1), and (R6,R1); (R6, R1) is not shown in Figure 6 because it is subsequently replaced by (P6,R1) when R6 is removed. `ConstructCDS` pops P4 from `CDStack`, creates 'while exit' node R7 (not shown in the figure) as a control dependence successor of P4 with label 'F', adds R7 to `Active.cfunreslist`, and pops 'while header' node R1. Next, the algorithm creates node S11, and resolves node R7 by creating control flow edge (R7,S11). Finally, `ConstructCDS` encounters *end pgm* in `AST`, so it connects the 'exit' node into `CDS`. `ConstructCDS` then finds that R6 and R7 have no control dependence children, so it removes them from `CDS`, and in adjusting control flow edges, adds edges (P4,S11) and (P6,R1) with label 'F' to `CDS`, and removes edges (R6, R1) and (R7, S11) from `CDS`.

3.1.2 Handling Structured Transfers of Control

Structured transfers of control have complex effects on control dependencies. Consider what occurs, for example, in Fragment A of Figure 7, where a *continue* is enclosed in the *if-clause* of *if-else* statement S2, such that the *if-clause* is executed when predicate P2 is true. When P2 is true, control flows back to S1. However, when P2 is false, statements after the *if-else* (S4 in the figure) are executed. Thus, statements after the *if-else* statement are control dependent on P2. Ballance and Maccabe[6] present the concept of follow regions to account for such control dependencies. A *follow region* summarizes the control dependencies for statements that appear after a compound statement. For example, the code after *if-else* statement S2 in Fragment A constitutes a region that follows the *if-else* control structure and is control dependent on P2-false. The use of follow regions also applies to nested statements. For example, Fragment B of Figure 7 shows a *while* loop that encloses *if-else* statement S3. Statement S5, the statement that follows S3, is control dependent on P3-false because the *if-else* statement contains *continue* statement S4; thus, the follow of S3 is P3-false. Statement S6, the statement that follows S2, is control dependent on P2-false or P3-false; thus, the follow of S2 is P2-false or P3-false.

We can use the concept of follow regions to compute control dependencies from the AST. However, as presented by Ballance and Maccabe, the concept is not sufficiently well-defined to let us calculate control dependencies correctly in all cases. For example, Ballance and Maccabe's approach does not compute control dependencies correctly for loops that contain *return* or *exit* statements. Consider Fragment C of Figure 7, in which the *while* loop contains an *exit* statement. In this case, Ballance and Maccabe's algorithm computes the control dependencies for S7 as "P1-false or P2-false". According to the traditional definition of control dependence[13], however, S7 is control dependent only on P1-false: there is no path from P2 to S7 on which every node other than P2 and S7 is postdominated by S7.

Ballance and Maccabe's approach also may calculate incorrect control dependencies for statements that follow *while* loops that contain *break* statements. For example, if the *exit* statement in Fragment C of Figure 7 were replaced by a *break* statement, Ballance and Maccabe's approach would still calculate that statement S7 is control dependent on "P1-false or P2-false". However, S7 postdominates P1, and every statement in the *while* loop; its control dependence conditions are identical to those for statement S1.

A third problem with Ballance and Maccabe's approach is that it does not address the problem of computing follow regions for control structures that contain multiple structured transfers of control. Such structures require special handling. For example, consider Fragment D of Figure 7, which is identical to Fragment C, except that S4 is now a *break* statement. As just discussed, if *exit* statement S5 were the only transfer of control out of the loop, S7 would be control dependent on P1-false. Alternatively, if *break* statement S4 were the only transfer of control out of the loop, S7 would not be control dependent on any statements in the loop. Consider what happens, however, when the *break* and *exit* statements are both present. In this case, S7 does not postdominate P1 or P3, but there are paths from P1 and P3 to S7 such that S7 postdominates all nodes on the paths, so S7 is control dependent on P1-false or P3-true. However, S7 is not control dependent on P2: on every path from P2 to S7 there is some node that is not postdominated by S7. It is not clear how Ballance and Maccabe's approach will calculate control dependencies in this case.

Fragment B of Figure 7 also illustrates a fourth insufficiency in Ballance and Maccabe's approach, which occurs when statements have multiple control dependencies. In the fragment, S6 is control dependent on P2-false or P3-false. However, by definition [13], a statement node in a control dependence graph can be a control dependence successor of only one region. If a statement is control dependent on multiple predicates, a *summary* region, that summarizes these control dependencies, is required. For Fragment B, a summary region R is created and made control dependent on P2-false and P3-false, and S6 is inserted into the graph as a child of R. It is not clear from the discussion in Reference [6] how Ballance and Maccabe's algorithm creates these summary regions.

`ConstructCDS` uses the concept of follow regions to construct CDSs for procedures that contains structured control transfer statements. However, `ConstructCDS` computes control dependencies that agree with those computed by the traditional algorithm [13], and does not exhibit the inadequacies of Ballance and Maccabe's approach. We next discuss the way in which `ConstructCDS` computes control dependencies for control structures that contain single transfers of control.

Single structured transfers of control

To compute the control dependencies for statements that follow a control structure S that contains a single structured transfer of control, T, we use information about sequences of predicate conditions from P_n, the predicate associated with S, to P_1, the predicate that controls T. We call such a sequence of predicates a *predicate path*,[8] and define it as follows:

> **Definition 1:** A *predicate path* is a sequence $\Pi = \{P_n, \ldots, P_1\}$ such that each P_i, $(1 \leq i \leq n)$ represents (1) a predicate that controls a conditional statement and (2) a true or false branch from that predicate, such that each P_i $(1 \leq i < n)$ is control dependent on P_{i+1}, and such that if P_i and P_j $(i \neq j)$ are both members of Π, then $P_i \neq P_j$.

A predicate path describes a chain of control dependence conditions that lets flow of control reach a transfer of control embedded in a structure, starting from the predicate that controls that structure and ending at the predicate that controls the transfer of control. We are also interested in the control dependence conditions that let us avoid reaching such a transfer of control. Toward this end, we define the concept of control dependence negation[9] as follows:

> **Definition 2:** Let $\Pi = \{P_n, \ldots, P_{wp}, \ldots, P_1\}$ be a predicate path such that P_{wp}, $1 \leq wp \leq n$, is the first *while* predicate in the sequence from P_n to P_1. The *control dependence negation* of Π, denoted $\neg\Pi$, is the disjunction:

$$\begin{array}{l} \quad P_n \ \wedge \ P_{n-1} \ \wedge \ \ldots \ \wedge \ P_{wp-1} \ \wedge \ \neg P_{wp} \\ \vee \ P_n \ \wedge \ P_{n-1} \ \wedge \ \ldots \ \wedge \ \neg P_{wp-1} \\ \qquad\qquad\qquad\quad \ldots \\ \vee \ \neg P_n \end{array}$$

To relate the control dependence of the statement that immediately follows a control structure to the predicate that controls that structure, we use the following lemma:[10]

> **Lemma 1:** Let S be a control structure controlled by predicate P. Suppose that S has a single control transfer statement T nested within it, such that T is control dependent on P with label L. Let S.follow be the region that summarizes the control conditions that hold after S is executed, and suppose the target of T postdominates S_1, the first statement in S.follow. Then S_1 is control dependent on P with label $\neg L$.

> **Sketch of Proof:** If S contained no embedded transfers of control, then S_1 would postdominate all of S. However, because S contains a transfer of control, T, the path in the control flow graph from T does not reach S_1 when the edge from P with label L is taken. Thus, S_1 does not postdominate statements that are control dependent on P with label L. However, S_1 postdominates any statement control dependent on P with label $\neg L$, because any path from S through the edge labeled $\neg L$ to the end of the program goes through S_1. Therefore, S_1 is control dependent on P with label $\neg L$. □

[8] Our definition of predicate path is similar to that of Reference [6].

[9] Ballance and Maccabe[6] also define control dependence negation. However, their definition is inadequate in certain cases; our definition corrects theirs.

[10] Lemma 1 is restated from Reference [6] with modifications. We use control dependence instead of immediate control dependence in keeping with the traditional definition of control dependence [13], i.e., a node cannot postdominate itself. We also relate the control dependence of the first statement in S.follow to the postdominance relation with respect to the target of the transfer of control.

```
begin program
...
    if (P)
      ...
      <exit statement>
      ...
    endif
    S1
...
end program
```

Fragment E

```
begin program
...
    while (P)
      ...
      <exit statement>
      ...
    endwhile
    S1
...
end program
```

Fragment F

Figure 8: Code fragments that illustrate the implications of Lemma 1.

To understand the implications of Lemma 1, consider Figure 8. Fragment E contains an *if* statement with an embedded transfer of control that is control dependent on P-true. Regardless of the type of this transfer of control (i.e., *continue*, *break*, or *return*), the target of the transfer of control postdominates S_1. Thus, by Lemma 1, S_1 is control dependent on P-false. Fragment F contains a *while* statement with an embedded transfer of control that is control dependent on P-true. If the transfer of control is either a *continue* or a *break*, all paths through P reach S_1, and thus, S_1 is not control dependent on P. Lemma 1 handles this case because of its requirement that the target of the transfer of control postdominate S_1. If the transfer of control is a *continue*, the target of this transfer of control is the header of the *while* loop, which does not postdominate S_1. Thus, because the hypothesis of Lemma 1 does not hold, Lemma 1 does not apply. Likewise, if the transfer of control is a *break*, the target of this transfer of control is S_1, which does not postdominate itself. Again, because the hypothesis of Lemma 1 does not hold, Lemma 1 does not apply. However, if the transfer of control is a *return* or *exit*, then its target is the end of the program, which does postdominate S_1, and thus, by Lemma 1, S_1 is control dependent on P-false.

We use Lemma 1 to prove the following Theorem, which relates the control dependence conditions for statements that follow a control structure to the predicate path from the control structure to an embedded transfer statement, for structures that contain only single structured transfers of control:[11]

> **Theorem 1:** Let S be a control structure controlled by predicate P_i. Suppose that S has a single transfer of control statement T embedded within it, such that $\Pi = \{P_i, \ldots, P_1\},\ i \geq 1$ is the predicate path from P_i to T. Suppose the target of T postdominates the first statement, S_1, in S.follow.
>
> Then S_1 is control dependent on $\neg\Pi$.
>
> **Sketch of Proof:** Induction on the length of Π. Let $\Pi = P_i$. Because the transfer of control in S is controlled by P_i, it follows from Lemma 1 that S_1 is control dependent on $\neg\Pi = \neg P_i$. Suppose that Theorem 1 holds for predicate paths Π of length n (where $\Pi = \{P_n, \ldots, P_1\}$), and suppose that T is controlled by predicate path $P_{n+1} \cdot \Pi = \{P_{n+1}, P_n, \ldots, P_1\}$. There are two cases.
> *Case 1:* P_{n+1} is a predicate that controls an *if* statement. S_1 is reached if T is not reached, which occurs when S is not entered or when S is entered and T is not reached. But this is just $P_{n+1} \wedge \neg\Pi \vee \neg P_{n+1}$, which is $\neg(P_{n+1} \cdot \Pi)$. Thus, S_1 is control dependent on $\neg(P_{n+1} \cdot \Pi)$.
> *Case 2:* P_{n+1} is a predicate that controls a *while* statement. Clearly, S_1 is reached if $\neg P_{n+1}$.

[11] Theorem 1 is also restated from Reference [6]. However, our definition of Π differs.

Consider the case when S is entered. Every path through S that avoids T reaches the *while* header where it again enters S, and could reach T. Thus, S_1 does not postdominate any statements that are control dependent on P_{n+1}, and S_1 is only control dependent on $\neg P_{n+1}$, which is $\neg(P_{n+1} \cdot \Pi)$ because P_{n+1} is the first *while* predicate in the predicate path. □

Theorem 1 tells us that, given a control structure S controlled by predicate P, if S contains a single structured transfer of control T, we can compute the control dependencies for statements that follow S by calculating the predicate path Π from P to T, and calculating $\neg\Pi$. `ConstructCDS` performs this computation.

`ConstructCDS` uses `TransferList` to record the path from a predicate to an embedded control transfer statement. Whenever the algorithm creates a region node, it records, in `RegionTable`, the predicate path that represents the chain of control dependencies from the procedure entry to the region. When `ConstructCDS` encounters a transfer of control, it records the predicate path associated with that transfer, as listed with the `Active` node, in the `TransferList` entry associated with the topmost predicate node on `CDStack`. On reaching the end of a structure, the algorithm uses the `TransferList` associated with that structure's predicate node to compute control dependencies for code that follows the structure. The algorithm then passes `TransferList` information down, as necessary, to the next enclosing predicate, for use when the end of the enclosing structure is encountered.

We now describe the operations of `ConstructCDS` for cases where single transfers of control are involved. When `ConstructCDS` encounters a *continue* statement, it creates a node for that statement, and adds it to the CDS as a child of `Active`. The algorithm uses `CDStack` to determine the type of control structure that immediately encloses the *continue* statement. *Continue* statements cannot affect the control dependencies for statements that follow a 'while' loop, so if the enclosing structure is a *while* loop, no action is necessary. However, if the enclosing control structure is an *if-else* predicate, `ConstructCDS` notes the predicate path associated with `Active`, and records it on `TransferList[N].continue`, where N is the predicate node associated with the enclosing control structure. *Continue* statements also induce control flow effects that must be modeled by explicit edges in the CDS. To account for these effects, `ConstructCDS` inserts explicit control flow edges from nodes that represent *continue* statements to their enclosing 'while header' nodes.

When `ConstructCDS` encounters a *break* statement, it creates a node for that statement, and adds it to the CDS as a child of `Active`. A *break* statement, like a *continue* statement, can affect the control dependencies for statements that follow any *if-else* statements that enclose it. Thus, `ConstructCDS` notes the predicate path associated with `Active`, and records it on `TransferList[N].break`, where N is the predicate node associated with the enclosing control structure. To capture control flow effects caused by *break* statements, the algorithm also adds nodes that represent *break* statements to a list on the enclosing 'while header' node in `LoopStack`. Later, when `LoopStack` is popped, `ConstructCDS` adds these break statements to the `.cfunreslist` on the parent node of the 'while header' node, ensuring that control flow edges are added to the next node that is created.

When `ConstructCDS` encounters a *return* or an *exit* statement, it follows the same procedure as for *break* statements, and it also records the predicate path associated with `Active` on `TransferList[N].return-exit`, where `N` is the predicate node associated with the enclosing control structure. In this case, to handle control flow effects, the algorithm immediately creates explicit control flow edges to the 'exit' node from nodes that represent *return* or *exit* statements.

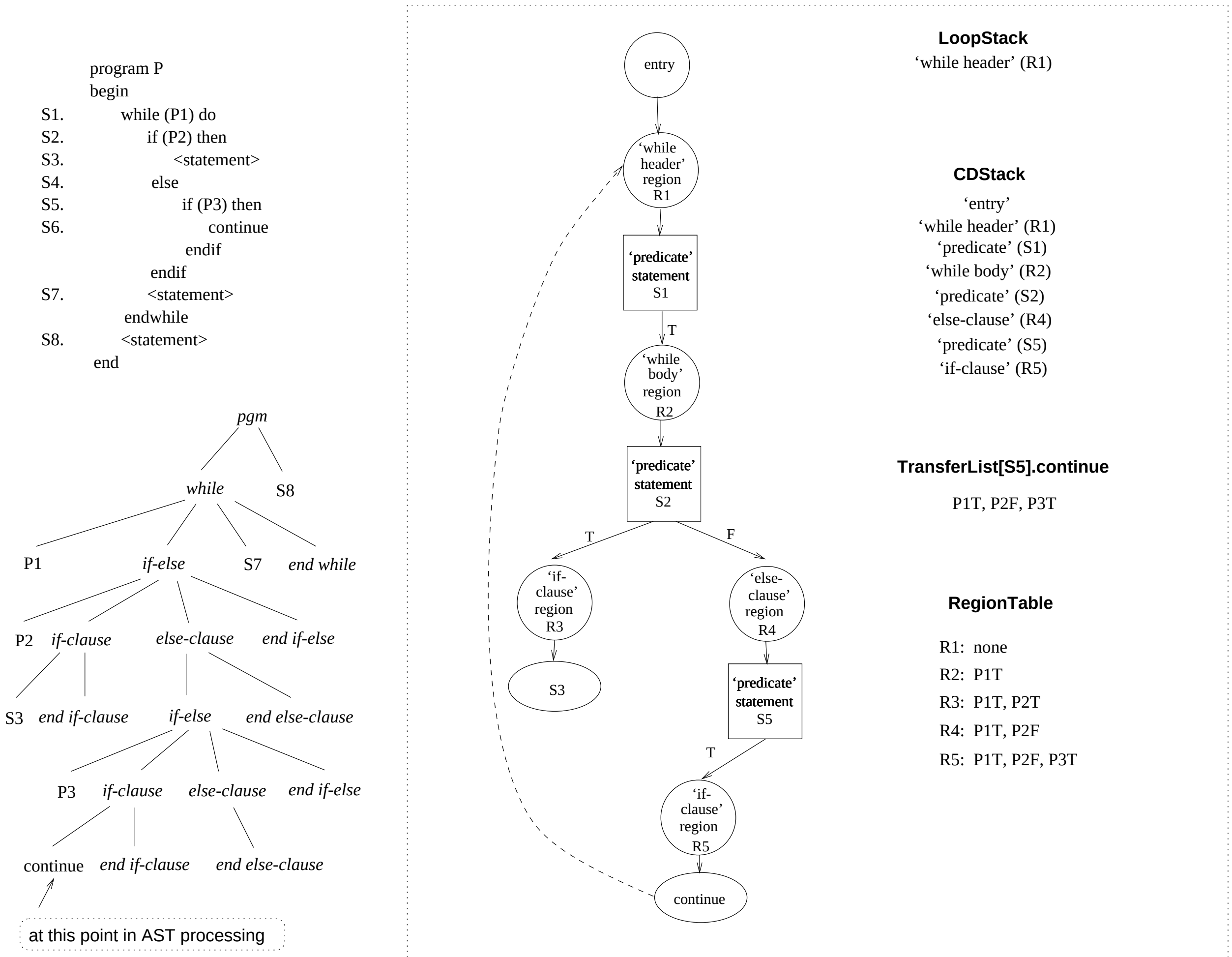


Figure 9: Snapshot of the CDS and stacks just after processing a *continue* statement that is nested in control structures. The partial AST is shown on the left, and the portion of the CDS that has been constructed at this point is in the center.

To illustrate the way in which `ConstructCDS` handles single transfers of control, Figure 9 shows a snapshot of the CDS construction for a program P after `ConstructCDS` has processed a *continue* statement. The figure also shows the states of `CDStack`, `LoopStack`, `RegionTable`, and `TransferList` at that point in the processing. The predicate path associated with the *continue* is calculated by locating the path attached to its parent region node in `RegionTable`. This path is then saved in `TransferList[S5].continue`. For this example, other types of transfer statements result in the same state, except that the predicate path would be saved on the field of the `TransferList` that is appropriate for the type of transfer.

When `ConstructCDS` encounters the end of a control structure, it uses `ComputeFollow` to perform two tasks. First, `ComputeFollow` uses the predicate paths on `TransferList` to determine `Follow` – the region or regions that summarize the control dependence conditions for statements that follow the control structure. The algorithm uses `Follow` to determine the new active region, and pushes it onto `CDStack` and adjusts the `.cfunreslist` appropriately. Second, `ComputeFollow` determines whether or not the transfers of control that are embedded in the current control structure can affect the statements that follow the enclosing control structure. If a transfer of control can affect the control dependencies of statements that follow the enclosing control structure, the algorithm adds the predicate paths associated with the transfer to the `TransferList` for the enclosing structure.

For example, when `ConstructCDS` encounters an *end if-else* whose `TransferList` indicates that is has a transfer of control nested within it, `ConstructCDS` calls `ComputeFollow`, with the *if-else* predicate node as a parameter. `ComputeFollow` considers the predicate path in `TransferList[N].continue`, `TransferList[N].break`, or `TransferList[N].return-or-exit`, where `N` is the *if-else* predicate node. `ComputeFollow` extracts the portion of the predicate path that denotes the predicate path from `N` to the transfer statement, takes the control dependence negation of that portion of the predicate path, and assigns the result to `TempCond`. The algorithm then determines the region node in the CDS that corresponds to the predicate path listed in `TempCond`, and assigns this node to `Follow`. Statements that follow the *if-else* are control dependent on that region node; `ConstructCDS` sets `NewActive` to this node, associating `Active.cfunreslist` with this new active node. Actions performed when `TempCond` contains more than one node occur only when multiple transfers of control are present; we discuss these actions in the next section.

When called to compute follow information at the end of a structure N, to update `TransferList` for enclosing control structures, `ComputeFollow` considers `ParentPredicate`, the predicate associated with the control structure immediately enclosing N. If `ParentPredicate` represents an *if-else* statement, then the algorithm places the predicate path listed on `TransferList[N].continue`, `TransferList[N].break`, or `TransferList[N].return-or-exit` onto the corresponding list on `TransferList[ParentPredicate]`. If `ParentPredicate` is a *while* statement then *continue* and *break* information do not need to be passed to it from N, for single transfers of control, because *continue* and *break* statements cannot affect the control dependencies of statements following the *while* loop (*break* statements must be passed to `TransferList[ParentPredicate]` if there is a *return* or *exit* present; we discuss this further when we consider multiple transfer of control statements). However, the presence of a *return* or *exit* statement in a structure, indicated by a nonempty `FollowArray[N].return/exit`, can affect statements that follow an enclosing *while* structure: the statements that follow the *while* loop structure are control dependent on not entering the loop. Thus, when `ParentPredicate` is a *while* statement, `ComputeFollow` passes the predicate path listed on `TransferList[N].return-or-exit` to the corresponding list on `TransferList[ParentPredicate]`.

When `ConstructCDS` encounters an *end while* AST node for a *while* statement that encloses a *return* or an *exit* statement, it computes the follow information, `Follow`, for the *while* statement using the `TransferList` associated with the *while* predicate. This computation is similar to the computation just described for the *end if-else* statement, except that on *end while*, the algorithm need not consider embedded *continue* statements, because they cannot affect the control dependencies for statements that follow the *while* statement.

Multiple structured transfers of control

We next consider the way in which multiple transfers of control affect the control dependence conditions for statements that follow a control structure. We require the following definitions:

Definition: Let $\Gamma = \{\Pi_1, \Pi_2, \ldots, \Pi_m\}$ be a set of predicate paths. The *control dependence disjunction* of Γ, denoted by $\oplus\Gamma$, is the disjunction $\Pi_1 \ \vee\ \Pi_2 \ \vee\ \cdots\ \vee\ \Pi_m$.

Definition: Let $\Gamma = \{\Pi_1, \Pi_2, \ldots, \Pi_m\}$ be a set of predicate paths. The *control dependence conjunction* of Γ, denoted by $\otimes\Gamma$, is the conjunction $\Pi_1 \ \wedge \ \Pi_2 \ \wedge \ \cdots \ \wedge \ \Pi_m$.

The following theorem relates the control dependence conditions for statements that follow a control structure to predicate paths, for structures that contain multiple structured transfers of control:

Theorem 2: Let S be a control structure controlled by predicate P. Suppose that S has embedded within it m transfers of control T_i $(1 \leq i \leq m)$ controlled by predicate paths Π_j $(1 \leq j \leq n, \ m \leq n)$. Suppose that S_1 is the first statement in S.follow. The effects of transfer statements on the control dependencies for statements in $S.follow$ are determined by the following three cases:
Case 1: If for all T_i, the target of T_i does not postdominate S_1, the transfers of control have no effect on the control dependence of S_1.
Case 2: If there exists some transfer of control T_i $(1 \leq i \leq m)$ such that the target of T_i postdominates S_1, and for all T_i in which the target of T_i does not postdominate S_1 their target is not S_1, then S_1 is control dependent on $\otimes$A, where A $= \{\neg\Pi_i | \Pi_i$ is associated with some T_i whose target postdominates $S_1\}$.
Case 3: If there exists some transfer of control T_i $(1 \leq i \leq m)$ such that the target of T_i postdominates S_1, and there exists a transfer of control T_j $(1 \leq j \leq m, \ i \neq j)$ such that the target of T_j is S_1, then S_1 is control dependent on $\otimes$A $\vee$ $\oplus$B, where A $= \{\neg\Pi_i | \Pi_i$ is associated with some T_i whose target postdominates $S_1\}$ and B $= \{\Pi_j | \Pi_j$ is associated with some T_j whose target equals $S_1\}$

Sketch of Proof:

We first show that the three cases account for all possible combinations of transfer of control statements. A control structure S can contain three types of transfer statements, distinguished in terms of their postdominance effects, that may affect the control dependence for S_1, the first statement in S.follow: (1) the target of the transfer may neither postdominate S_1 nor be S_1; (2) the target of the transfer may be S_1; and (3) the target of the transfer may postdominate S_1. To see examples of these three types of transfer statements, consider Figure 10.

```
S: if (P)
     ...
       while (P)
         ...
         continue
         ...
       endwhile

     ...
     endif
     S1
```

Fragment A

```
S: while (P)
     ...
       if (P)
         ...
         break
         ...
       endif

     ...
     endwhile
     S1
```

Fragment B

```
S: if (P)
     ...
       while (P)
         ...
         exit
         ...
       endwhile

     ...
     endif
     S
```

Fragment C

Figure 10: Code fragments that illustrate cases of Theorem 2.

Fragment A in the figure gives an example of the first type of transfer statement. In this case, the *continue* is a transfer statement in S, but the target of the *continue*, the *while* header, does not postdominate S_1 and the target of the *continue* is not S_1. Fragment B in the figure gives an example of the second type of transfer statement. In this case, the target of the *break* statement is S_1, which is the first statement after the enclosing *while* loop. Finally, Fragment C in the figure gives an example of the third type of transfer statement. In this case, the target of the *exit* statement is the end of the program, which postdominates S_1.

Because control structures can contain multiple transfer statements, we must consider combinations of types of transfer statements. There are seven possibilities, excluding the case where no statements are transfer statements. These combinations are summarized in Table 1, where the first column numbers the possibilities, the next three columns indicate the three types of transfer statements, and the last column indicates the case that applies to that combination of types.[12]

	type (1)	type (2)	type (3)	case that applies
1	X			Case 1
2		X		Case 1
3			X	Case 2
4	X	X		Case 1
5	X		X	Case 2 - ignore type (1)
6		X	X	Case 3
7	X	X	X	Case 3 - ignore type (1)

Table 1: Possible combinations of types of transfer statements and the case that handl es them.

Case 1: The transfers do not affect dependencies; there is nothing to handle for this case.
Case 2: Using Theorem 1, for each Π_i, we can get $\neg\Pi_i$, the control dependence conditions for reaching S_1 when the T_i are considered separately. To reach S_1, we must avoid T_i for all $1 \leq i \leq m$. But this means that S_1 is reached if $\neg\Pi \wedge \cdots \wedge \neg\Pi_m$, which is just $\otimes$A.
Case 3: Using Theorem 1, we can get $\neg\Pi_i$ for each Π_i, which are the control dependence conditions for reaching S_1 when the T_i are considered separately. We can also get Π_j, which are the control conditions for reaching S_1 from a *break* statement. To reach S_1, we must avoid T_i, which is just $\otimes$A where A $= \Pi_{i_1}, \Pi_{i_2}, \ldots, \Pi_{i_k}$ Another way to reach S_1 is to reach some T_j where the target of T_j is S_1, which is just $\oplus$B. Thus, S_1 is reached if $\otimes$A or $\oplus$B, which is just $\otimes\text{A} \vee \oplus\text{B}$.
□

The formulas given by Theorem 2 provide a mechanism for calculating control dependencies in the presence of multiple structured transfers of control. `ConstructCDS` records predicate paths for each transfer it encounters in `TransferList`. Aside from the need to record multiple predicate paths in fields of `TransferList`, the algorithm otherwise handles multiple transfer statements similarly to single transfers. The principle difference in the algorithm occurs in the `ComputeFollow` procedure, where the algorithm calculates dependencies for code that follows control structures.

When `ConstructCDS` encounters an *end if-else* statement, `ComputeFollow` computes follow information for the *if-else* statement N using the `TransferList`s associated with N. The algorithm considers all predicate paths in `TransferList[N].continue`, `TransferList[N].break`, and `TransferList[N].return-or-exit`. `ComputeFollow` extracts the portions of those paths that correspond to paths from N to the transfer statements, calculates the control dependence negations of those path portions, then calculates the control dependence conjunction of those negations, and assigns the result to `TempCond`. The algorithm then determines the region nodes in the `CDG` that correspond to the conditions listed in `TempCond`, and uses these nodes to find a new active region or create a summary region, that encodes the control dependencies for statements that follow the control structure.

Next, to account for the effects of transfers on structures that enclose N, `ComputeFollow` passes predicate paths listed on `TransferList[N]` to the `TransferList`s associated with the enclosing control structure. If the enclosing structure ParentPredicate is a *while* statement, and there are *return* or *exit* statements, but

[12] Theorem 2 subsumes Theorem 1: single transfers of control are accounted for by cases 1 and 2. We presented Theorem 1 separately to simplify the presentation and proofs.

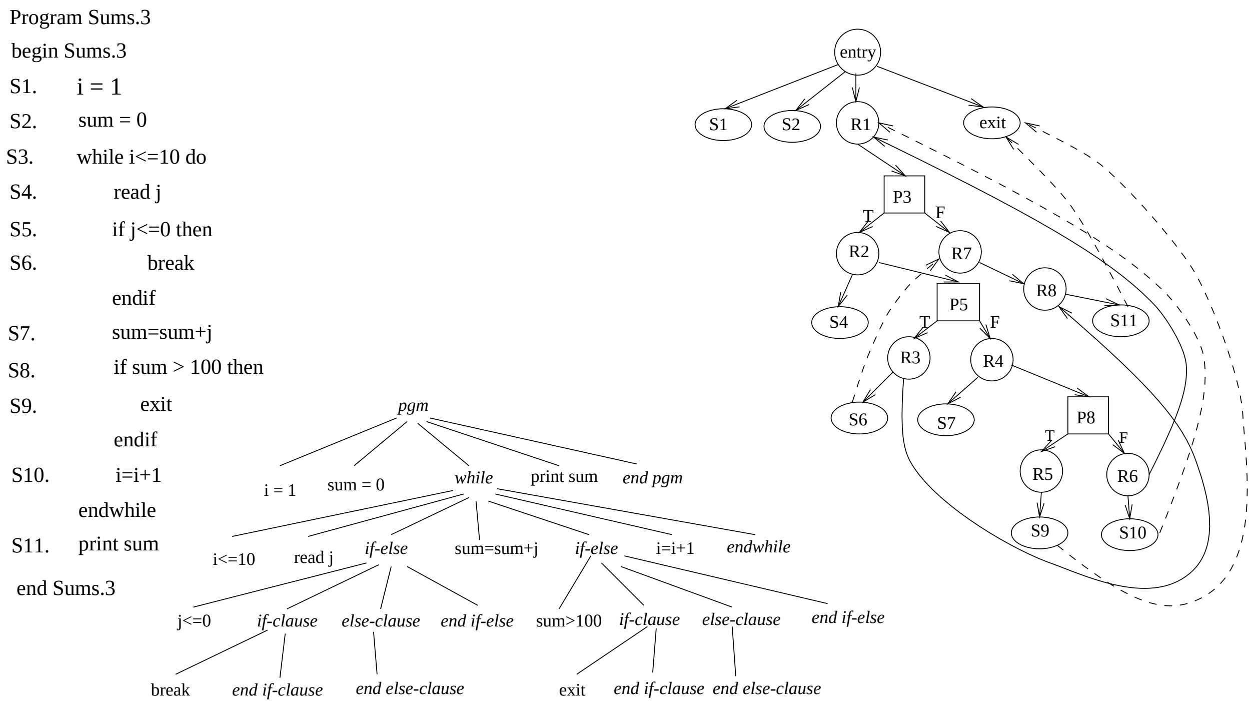


Figure 11: Program `Sums.3` with structured transfers of control, its abstract syntax tree and its CDS (we omit the DDS). The dashed lines in the CDS represent explicit control flow edges that our algorithm computes during construction of the CDS; other control flow edges are implicit due to ordering of nodes in the CDS.

---

no *break* statements, on the TransferList for N, `ComputeFollow` passes predicate paths for *return* and *exit* transfers to `TransferList[ParentPredicate].return-or-exit`. If there are *break* statements in the TransferList at N, their predicate paths are also passed to `TransferList[ParentPredicate].break`.

When `ConstructCDS` encounters an *end while* AST node for a *while* that encloses *return* or *exit* statements, it computes the follow information, `Follow`, for the *while* statement using the `TransferList` associated with the *while* predicate. This computation is similar to the computation for the *end if-else* statement, except that for *end while*, the algorithm need not consider embedded *continue* statements, because they cannot affect the control dependencies for statements following the *while* statement.

Example

We next consider an example that illustrates the way in which `ConstructCDS` handles procedures that contain multiple structured transfers of control. Figure 11 displays such a program, `Sums.3`, with its AST and CDS. We trace the algorithm's execution beginning with its processing of the *break* statement. At this time, nodes 'entry', S1, S2, R1, P3, R2, S4, P5, and R3 have been added to `CDS`. `CDStack` contains 'entry', R1, P3, R2, P5, and R3, `LoopStack` contains R1, `RegionTable` contains predicate paths for R1, R2, and R3, and `TransferList` is empty. When `ConstructCDS` encounters the *break* statement, it creates 'statement' node S6 in `CDS` as a child of the `Active` node, R3. The algorithm places predicate path P3T,P5T onto `TransferList[P5].break`. The algorithm also places S6 on `LoopStack[R1].breaklist` so that when it encounters the end of the loop, it can resolve the control flow from S6 to the statement that follows the loop.

Next, `ConstructCDS` encounters the *end if-clause* AST node, and pops the top entry from `CDStack`. On encountering the *else-clause*, the algorithm creates R4 in `CDS` as a control dependence successor of P5 with label 'F', adds R4 to `RegionTable` with predicate path P3F, P5F, and pushes R4 onto `CDStack`. Because there are no statements in this *else-clause*, the algorithm next encounters the *end else-clause* AST node, and pops R4 from `CDStack`.

Next, `ConstructCDS` encounters the *end if-else* AST node. The algorithm pops P5 from `CDStack`, and because `TransferList[P5].break` is nonempty, calls `ComputeFollow` with P5. `ComputeFollow` performs the usual calculations, and determines that code following the *if-else* is control dependent on P5-false; thus the algorithm pops region R2 from `CDStack` and places R4 on the stack in its place. Before returning, `ComputeFollow` passes `TransferList[P5].break` to `TransferList[P3].break`.

Next, `ConstructCDS` adds S7, P8, and R5 to `CDS`, adds R5 to `RegionTable` with predicate path P3F,P5F,P8T, and pushes P8 and R5 onto `CDStack`. When `ConstructCDS` encounters the *exit* statement, it creates statement node S9, and assigns predicate path P3T,P5F,P8T to `TransferList[P8].return-or-exit`. The algorithm also creates control flow edge (S9,'exit').

Subsequently, `ConstructCDS` encounters the *end if-clause* for the *if-else* at S8, and pops R5 from `CDStack`. The algorithm then processes the empty *else-clause* as it processed the previous one. Next, the algorithm encounters the *end if-else*, and calls `ComputeFollow` with P8. `ComputeFollow` pops R4 from `CDStack` and replaces it with R6, the region that represents the control dependencies for statements that follow *if-else* statement S8.

The algorithm next adds S10 to `CDS` as a child of R6, and then encounters the *end while* AST node. `ConstructCDS` pops R2 from `CDStack`, resolves the control flow edge from S10 to R1, and pops P3 from `CDStack`. Then, `ConstructCDS` adds 'while exit' region node R7 to `CDS` and lists this node as unresolved.

Next, the algorithm must account for the effects of *break* statements contained within the *while* statement: the algorithm creates a control flow edge from S6 to the 'while exit' node. (Additional actions associated with `goto` statements are discussed in the next section.) The algorithm then calculates control dependence effects caused by *break*, *return*, or *exit* statements in the loop; this calculation prompts creation of control dependence edge (R6,R1). The algorithm then pops R1 from `CDStack` and from `LoopStack`.

The algorithm then calls `ComputeFollow` with P3. The algorithm calculates that code following the loop is control dependent on P5-true or P3-false, creates a summary node, R8, to encode these conditions, adds R8 to `CDS`, adds R8 to `RegionTable` with predicate paths P3T,P5T and P3F, pops `CDStack`, and pushes R8 onto it. Because there are no predicate nodes remaining on `CDStack`, the algorithm does not need to pass `TransferList` information to the `TransferList` of an enclosing predicate node.

Finally, the algorithm adds S11 to `CDS` as a child of `Active`, resolves control flow edges, and removes the 'entry' node from `CDStack`. The algorithm also connects the exit node in to `CDS`. Finally, `ConstructCDS` processes the nodes in `CDS` to remove spurious region nodes; it finds that R7 has only one child that is also a region, so it combines nodes R7 and R8. Because we wanted to illustrate the construction of summary regions, we illustrated the CDS in Figure 11 before regions R7 and R8 were combined.

3.1.3 Handling Goto Statements

Many *goto* statements in a program are used as multilevel *continues* or *breaks* [6]. In these cases, `ConstructCDS` constructs the CDS from the AST without requiring control flow information. To do this, the algorithm uses techniques that are similar to those used to handle *continue* and *break* statements. However, if during this processing, ConstructCDS determines that a *goto* statement is not used as a multilevel *continue* or *break*, it sets a flag, AdjustFlag, to indicate that the CDS may not contain correct control dependence information, and must later be adjusted. If AdjustFlag is set when the *end pgm* node is encountered in the AST, ConstructCDS calls procedure Adjust(), which computes exact control dependencies using the control flow graph that it constructed, and adjusts the CDS accordingly.

ConstructCDS uses a table, LabelTable, to record information about *goto* statements and *labels*. When ConstructCDS encounters a *goto* statement, it creates a statement node and adds it to the CDS as a child of Active. Using the label, Label, that is the target of this *goto* statement, the algorithm uses Search() to find the location of Label in LabelTable. If Label is found in LabelTable and LabelTable[location].region is nonempty, then the location of Label was previously encountered in the program. In this case, the *goto* is a backward transfer of control, with two possible interpretations that we consider. First, if the leftmost child of the target region of this *goto* is the 'while header' node on top of LoopStack, then the *goto* statement is a *continue*. In this case, ConstructCDS updates TransferList information within the enclosing *while* loop using steps that are similar to those used to process continue statements. Second, if the leftmost child of the target region of this *goto* is not a 'while header', we do not know, without control flow information, whether this *goto* represents a loop backedge or an unstructured transfer of control. Thus, ConstructCDS sets AdjustFlag so that the control dependencies will later be adjusted. In either case, because the target Label was located in LabelTable, the algorithm creates a control flow edge from the *goto* statement.

If Label is found in LabelTable but LabelTable[location].region is empty, or Label is not found in LabelTable, then the target of the *goto* statement has not been encountered, so the *goto* is a forward branch in the program. In this case, ConstructCDS assumes this *goto* is a *break*. After adding Label to LabelTable[Location].label and the *goto* to LabelTable[Location].gotolist, the algorithm updates TransferList information as if this statement were a *break* statement.

When ConstructCDS encounters a *label* in the AST, it creates a new region node, LabelNode, adds it to the CDS as a child of Active, and pushes it onto CDStack. The algorithm then searches for the label in LabelTable. If this label is not found, then it is added to LabelTable. Otherwise, this label has been previously referenced; control flow edges are created to LabelNode from all *goto* statements that have this label as their target.

In addition to the usual actions taken by ConstructCDS when it detects an *end while* AST node, the algorithm considers the next node in the AST using a LookAhead to determine if any of the *goto* statements represented by the predicate paths in LoopStack[top].possiblebreaks are *break* statements for that loop. If it finds a *label* that is the target of any of these *goto* statements, it treats this *goto* statement as a *break* and adds information about it to TransferList[top].break; the *goto* statement is then processed with the other *break* statements in the loop. If the algorithm does not find a *label* that is the target of one of the *goto* statements represented by a predicate path in LoopStack[top].possiblebreaks, it assumes that the *goto* is a *break* for an enclosing *while* loop, and adds this *goto* information to the LoopStack[top-1].possiblebreaks. If there is no other enclosing *while* loop, the *goto* is not a *break*, and ConstructCDS sets AdjustFlag.

Consider, for example, what ConstructCDS does given the AST shown in Figure 12. The algorithm processes the first three nodes in the AST in the usual way to put 'entry', S1 and S2 in the CDS, and

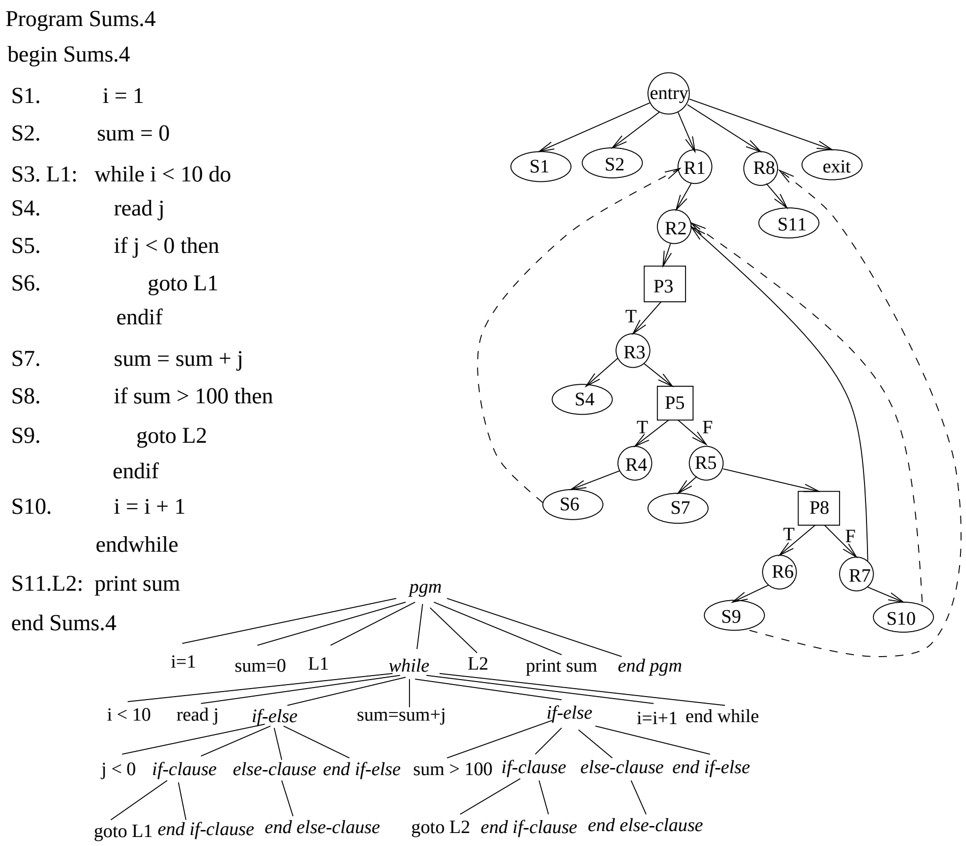


Figure 12: Program `Sums.4` with *goto* statements that behave like *continue* and *break* statements, its abstract syntax tree and its CDS (we omit the DDS). Dashed lines in the CDS represent explicit control flow edges that our algorithm computes during construction of the CDS; other control flow edges are implicit due to ordering of nodes in the CDS.

'entry' on `CDStack`. Then `ConstructCDS` encounters the *label* node L1, and adds R1 to `CDStack` to represent the 'label' region. The algorithm updates `LabelTable` because L1 has been encountered. The algorithm processes the *while* loop in the usual way, creating 'while header', 'predicate', and 'while body' nodes R2, P3, and R3 respectively, and adding R2 and R3, along with their predicate paths, to `RegionTable`; it also pushes these nodes onto `CDStack` and adds R2 to `LoopStack`. The next nodes encountered in the AST cause S4, P5, and R4 to be added to `CDS`, R4 and its predicate path to be added to `RegionTable`, and P5 and R4 to be pushed onto `CDStack`. `ConstructCDS` then encounters S6, a *goto* statement. The algorithm adds a statement node for S6 to `CDS`, and gets the `Label`, L1, that is the target of this *goto*. When `ConstructCDS` searches `LabelTable`, it finds L1, and because R1, the region associated with L1, has 'while header' R2 as its leftmost child, the algorithm processes this *goto* as if it were a *continue* statement. Thus, the algorithm updates `TransferList[P5]` using steps identical to those used to process a *continue* statement.

After `ConstructCDS` processes S6, it encounters the *end if-clause*, and pops R4 from `CDStack`. The algorithm then creates an empty *else-clause* region R5, and detects the *end if-else* node in the AST. At this time, `ConstructCDS` pops both P5 and R3 from `CDStack`, and uses `TransferList[P5]` to compute the follow for P5, which is region R5. Thus, R5 becomes `NewActive` and the algorithm pushes it onto `CDStack`.

Next, `ConstructCDS` adds S7 as a child of R5, and processes the second *if-else* statement, causing P8 to be added as a child of R5 and pushed onto `CDStack`. The algorithm adds R6 as the region for the 'if-clause', adds R6 and its predicate path to `RegionTable`, pushes R6 onto `CDStack`, and adds S9, the second *goto* statement, as a child of this region. This time, `ConstructCDS` assumes the *goto* statement is a *break* statement because the *goto* is a forward branch; `TransferList` information is updated accordingly.

Subsequently, the algorithm pops R6 and processes the empty *else-clause* region causing R7 to be created, added to `RegionTable` with its predicate path, pushed onto `CDStack`, and then popped from `CDStack`. Next, the end of the *if-else* is encountered and P8 and R5 are popped from `CDStack`. `ConstructCDS` computes that R7 is the follow region for the structure, and pushes R7 onto `CDStack` as `NewActive`. The next AST node, S10, is added to the CDS as a child of this region.

When `ConstructCDS` detects the *end while* AST node, it pops R7 and R2 from `CDStack`. Additionally, the algorithm determines that S9 acts as a *break* for this *while* loop, and processes it as a *break* statement. Thus, the algorithm adds control dependence edge (R7, R2) to the CDS. The algorithm also adds control flow edge (S10, R2) to the CDS. In processing the *end while*, `ConstructCDS` adds a 'while exit' region to the CDS; because this 'while exit' region is later removed, we omit it from the CDS.

When `ConstructCDS` encounters L2, it creates region node R8, adds it to the `RegionTable` with the appropriate predicate path, adds it to the CDS, pushes it onto `CDStack`, and creates control flow edge (S9,R8). Then, the algorithm adds S11 as a child of R8 in the CDS. When the algorithm encounters the *end pgm* AST node, it pops 'label' regions R8 and R1 from `CDStack`. Finally, the algorithm connects the 'exit' node to the CDS. When `ConstructCDS` removes spurious region nodes, it removes R1 and R8 from the CDS, and updates control dependence and control flow edges accordingly. Figure 12 shows the CDS before these regions are removed.

3.1.4 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` must be considered for adjustment. `Adjust()`, shown in Figure 13, takes as input a CDS, `G`, and returns an updated version of `G` with correct control dependencies. We denote the control dependencies for a node `N` in the original version of `G` by $\mathtt{CD_G(N)}$.

`Adjust()` uses control flow information in `G` to compute postdominator information and control dependencies for each node. We denote these control dependencies for node `N` by $\mathtt{CD_{CF}(N)}$ because this method dependencies requires control flow information. After control dependencies for nodes in `G` are computed, `Adjust()` visits each node `N` in `G` in program order. As each `N` is visited, `Adjust()` compares correct control dependencies with those present in `G`. If these dependencies differ, control dependencies `M` in $\mathtt{CD_{CF}(N)}$ but not in $\mathtt{CD_G(N)}$ are inspected; these represent control dependencies missed by `ConstructCDS`. If no region currently exists for `M` in `G`, then a region `R` is created and added to `G`, and `R` is marked for later ordering. `Adjust()` also considers those extra control dependencies in `G` that were added by `ConstructCDS`; control dependence edges to those regions are removed and the regions are marked for reordering.

After `Adjust()` visits each node `N`, it orders the nodes in marked regions using the control flow among the nodes in the region. Finally, `Adjust()` removes redundant control flow edges, and returns `G`.

To illustrate the processing of unstructured *goto* statements, consider program `Sums.5` shown in Figure

```
procedure   Adjust(G)
input       G : a CDS or CDS subgraph with CD_G(N), the set of nodes on which N is control dependent,
                computed using ConstructCDS
output      G : with exact control dependencies

begin
    compute exact CD_CF(N) for each N in G using control flow CF
    foreach statement or predicate node N in program order do
        if CD_G(N) ≠ CD_CF(N) then
            foreach node M in CD_CF(N) - CD_G(N) do (* regions missed by ConstructCDS *)
                if there is no region node R for M then
                    create R for M; add R to table
                endif
                add edge R → N to G
                mark R for ordering
            endfor
            foreach node M in CD_G(N) - CD_CF(N) do (* extra regions created by ConstructCDS *)
                foreach region R summarizing M do
                    remove edge R → N
                    mark R for ordering
                endfor
            endfor
        endif
    endif
    foreach R that is marked do
        order nodes in R according to CF edges (* creates implicit control flow edges *)
    endfor
    remove redundant CF edges
    return G
end Adjust
```

Figure 13: Procedure `Adjust()` that converts `CDS` to a CDS with exact control dependencies.

14. `Sums.5` is semantically equivalent to `Sums.4`, shown in Figure 12, except that it is implemented with *if-else* and *goto* statements instead of *while* statements. `ConstructCDS` processes S1, S2, and S3 in the usual way yielding entry, S1, S2, R1, P3, and R2 in `CDS`, R1, R2 and their associated predicate paths in `RegionTable`, and entry, R1, P3, and R2 on `CDStack`. When the algorithm encounters S4, it creates node S4 as a child of R2 in `CDS`. Because S4 is a *goto* statement, the algorithm then uses `Search()` to locate the target of S4, L1, in `LabelTable`. `Search()` finds L1 in `LabelTable` with region R1, indicating that the *goto* is a backward branch. However, because the leftmost child of R1 is not a 'while header' node, the target of the branch is not a *while* loop. Thus, `ConstructCDS` sets `AdjustFlag`.

After the construction of the rest of the CDS shown in Figure 14, `ConstructCDS` calls `Adjust()` to compute correct control dependencies for nodes in the CDS; here, `G` is the CDS produced by `ConstructCDS`. In the first step, `Adjust()` uses the control flow information in `G` to compute control dependencies for each node in `G`. `Adjust()` then considers each node in `G` and finds that S5, P6, S8, P9, S11, and S12 have computed control dependencies that differ from those expressed by `G`. In particular, S5, and P6 are control dependent on P3-false and thus, are removed from region R1, and made control dependent on P3-false; P3-false is then 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, P9, S11, and S12. The resulting CDS is given in Figure 15.

```
Program Sums.5
begin Sums.5
S1.      i = 1
S2.           sum = 0
S3. L1:    if i > 10 then
S4.              goto L2
              endif
S5.          read j
S6.          if j < 0 then
S7.              goto L1
              endif
S8.         sum = sum + j
S9.         if sum > 100 then
S10.            goto L2
              endif
S11.       i = i + 1
S12.       goto L1
S13.L2:   print sum
end Sums.5
```

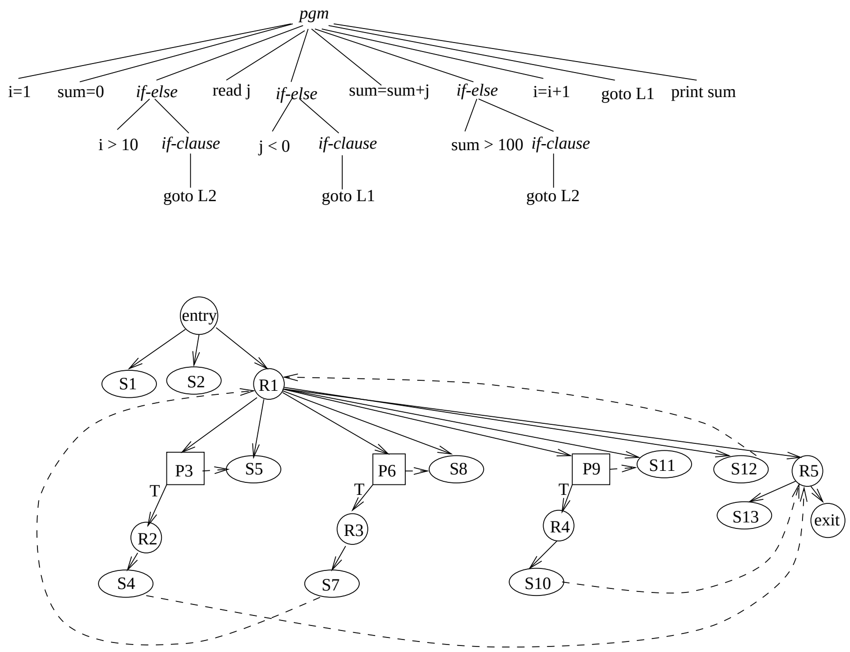


Figure 14: Program Sums.5 with *goto* statements, its abstract syntax tree, and its CDS (we omit the DDS). Dashed lines in the CDS represent explicit control flow edges that our algorithm computes during construction of the CDS; other control flow edges are implicit due to ordering of nodes in the CDS.

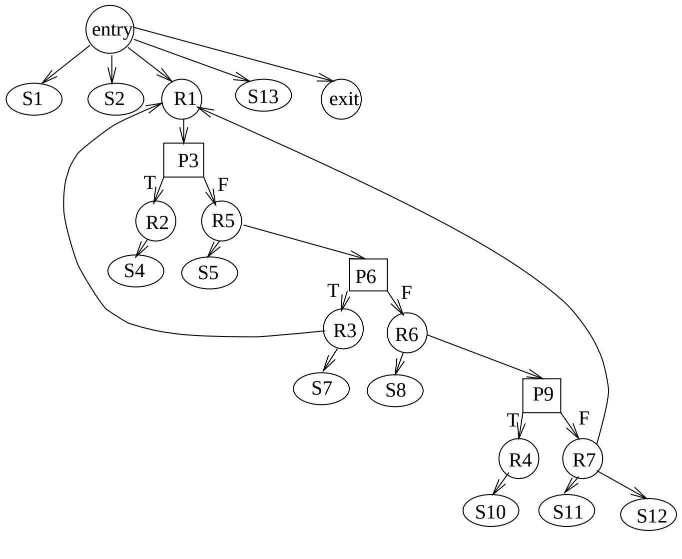


Figure 15: The CDS for the program in Figure 12 with correct control dependencies (control flow edges omitted).

3.1.5 Complexity

When programs are structured, the running time of `ConstructCDS` is linear in the number of statements in the program. When programs contain structured transfers of control, the running time of `ConstructCDS` also includes the time required to calculate follow information. This calculation is at worst proportional to the number of statements in the program. In the worst case, such a calculation could occur for each statement in the program, and therefore, the running time of the `ConstructCDS` algorithm for structured transfers is at worst quadratic in the number of statements in the program. In the case of unstructured transfers of control, the running time of `ConstructCDS` is dominated by the time required to adjust the CDS to represent exact control dependencies, which can take quadratic time in the worst case [10]. Thus, the running time of `ConstructCDS` for unstructured transfers of control is also quadratic in the number of statements in the program.

3.1.6 Improvements to `ConstructCDS`

As we mentioned above, to facilitate presentation, we have presented our algorithm as if it were traversing the AST. However, when implementing `ConstructCDS`, it is not necessary to create the entire AST before invoking the algorithm. Rather, the parser may serve as a driver for `ConstructCDS`, calling it with each AST node as the node is recognized.

Although we present our program dependence graphs with both implicit and explicit control flow edges, it may be more efficient to represent all control flow edges explicitly; our algorithm is easily modified to do this.

3.2 Constructing Data Dependence Subgraphs

After the CDS is constructed using the 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 each node that contains a definition of a variable to each node that contains a reachable use of that variable. *Anti-dependence* edges are added to the CDS from each node that contains a use of a variable to each node that contains a reachable definition of that variable. *Output-dependence* edges connect particular definitions of the same variable.

Traditional data flow analysis algorithms fall into two major categories: iterative algorithms and elimination algorithms [4]. Either of these techniques can be applied to our CDS to compute data flow information. We first discuss our adaptation of iterative algorithms to work on our CDS, and then discuss the adaptation of elimination algorithms to use our CDS.

Because our CDS also contains control flow information, we can adapt iterative data flow analysis algorithm, both forward and backward, to use it for the computation. Here, we describe our algorithm to compute *reaching definitions*, that is used to compute flow dependence information for the DDS; other data flow analysis algorithms are similar. To compute sets of definitions that reach each statement in the program, we first compute local definition information, and then we propagate it through the program using the implicit and explicit control flow edges in our CDS until the sets stabilize. Our algorithm, `ReachingDefs`, is given in Figure 16. `ReachingDefs` assumes that local data flow information, consisting of the usual `GEN`

```
procedure   ReachingDefs
input       CDS, control dependence subgraph, 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 (* compute data flow at node N *)
            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 16: Algorithm to compute reaching definitions using the CDS and control flow edges.

and KILL sets, has been computed and is attached to CDS nodes. The GEN set for a node consists of the definitions, if any, in that statement (node); the KILL set for a node consists of all other definitions of the GEN set's variable in the program. The GEN sets are easily computed as the CDS is built. Using the GEN sets, a KILL set is computed for each node. Predicate nodes and region nodes have neither GEN nor KILL sets.

To propagate data flow information, ReachingDefs uses 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. Because IN and OUT sets for a statement node may differ, we require both of these at each statement node in the CDS. However, because 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 one of two sets of equations depending on the node'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].

When programs are structured, elimination algorithms based on interval analysis [39, 43] can be adapted to use the CDS instead of the control flow graph [6]. In these cases, the control flow graph is not required in the analysis. However, if the program is highly unstructured or irreducible, these elimination algorithms may produce incorrect data flow information, and thus, these algorithms may require the use of iterative algorithms for some parts of the program. In this case, control flow information is required. An advantage of our CDS construction algorithm over that of Ballance and Maccabe [6] is that our algorithm computes control flow information as a program is parsed. Thus, our CDS can be used to correctly compute data flow information for all types of programs.

Program	Functionality
Bub.c	sorts a five integer array using bubble sort
Bisect.c	finds the square root of a number using interval bisection
Calc.c	implements a reverse Polish calculator described in Kernighan and Ritchie [30]
Euclid.c	implements Euclid's algorithm to compute the greatest common divisor of two integers
Find.c	partitions elements of an array so that all elements to the left of an indexed location are less than or equal to the value of the array at that index, and all elements to the right are greater than or equal to the value of the array at that index [26, 27]
Mid.c	finds the middle value of three integers
Search.c	uses binary search to find the location of an element or a location where new element should be inserted
Secant.c	computes the square root of a number using the secant method

Table 2: Descriptions of our subject programs.

4 Implementation and Empirical Studies

To investigate the efficiency and usefulness of `ConstructCDS`, we performed two empirical studies. In the first study, we compared the running time of a prototype implementation of `ConstructCDS` to the running time of an prototype implementation of a traditional CDS construction algorithm [13]. In the second study, we analyzed a number of C programs to determine how often our `ConstructCDS` algorithm could be used without requiring a call to `Adjust()`.

Study 1

Our `ConstructCDS` prototype takes a C source file, possibly containing multiple functions, as input. For each function F in the file, the prototype constructs the CDS and collects control flow information for F as the parser constructs the `AST` for F. The prototype also collects local data flow information during the parse, and attaches `GEN`, `KILL`, `DEF`, and `USE` sets to CDS nodes. At present, however, the prototype does not implement the `Adjust` procedure.

We implemented the algorithm described in Reference [13] as part of the `Aristotle` program analysis toolkit [24]; we refer to that implementation as the "FOW prototype". Like our `ConstructCDS` prototype, the FOW prototype takes a C source file, possibly containing multiple procedures, as input. For each function F in the source file, the prototype constructs the control flow graph for F during the parse. The FOW prototype also collects local data flow information during the parse, and attaches data flow sets to control flow graph nodes. After the parse is complete, the FOW prototype calculates postdominator information, computes control dependence information, and constructs the CDS, for each function in the source file.

Both prototypes output control flow, local data flow, and control dependence information to files. Both prototypes are written in C. We conducted this study on a Sun Microsystems SPARCstation[13] with 128 mb of virtual memory.

Our first study involved eight C programs of varying sizes. Table 2 gives a brief description of each of our subject programs. Because many of our subject programs contained small numbers of source lines, we duplicated procedure bodies to get "long" versions of the programs. Table 3 reports statistics on the programs in our test suite, along with their "long" versions. For each program and its "long" version, the table gives the number of functions in the program, the number of source statements, and the number of

[13] SPARCstation is a trademark of Sun Microsystems, Inc.

Program	Number of Procedures	Number of Source Statements	Number of CDS Regions
Bub.c	1	38	11
Bub.long.c	1	326	92
Bisect.c	1	34	10
Bisect.long.c	1	286	82
Calc.c	6	147	48
Calc.long.c	15	605	229
Euclid.c	1	15	4
Euclid.long.c	1	95	20
Find.c	1	66	21
Find.long.c	1	579	101
Mid.c	1	21	10
Mid.long.c	1	165	81
Search.c	1	33	10
Search.long.c	1	267	82
Secant.c	2	42	10
Secant.long.c	2	282	45

Table 3: Statistics about our subject programs. *Program*.c is the original version, and *Program*.long.c is the original program with one of the procedures duplicated ten times.

Program	ConstructCDS Prototype	FOW Prototype	FOW Prototype excluding postdominator calculation
Bub.c	0.7	0.7	0.6
Bub.long.c	1.8	16.7	2.6
Bisect.c	0.7	0.8	0.6
Bisect.long.c	1.0	7.5	2.0
Calc.c	1.3	1.5	0.9
Calc.long.c	1.5	24.2	5.9
Euclid.c	0.6	0.7	0.6
Euclid.long.c	1.0	1.4	1.0
Find.c	0.8	0.9	0.7
Find.long.c	1.0	70.1	8.1
Mid.c	0.6	0.7	0.6
Mid.long.c	1.0	11.0	1.6
Search.c	0.7	0.8	0.6
Search.long.c	1.0	12.3	1.8
Secant.c	0.7	0.8	0.6
Secant.long.c	2.0	7.1	3.1

Table 4: Times for computing the CDS for our subject programs. All times are user+sys time as reported by the Unix TIME command.

CDS regions. As an indication of the control dependence complexity of the programs, we also include in the table the number of CDS regions for each subject program.

To investigate the efficiency of our `ConstructCDS` algorithm, we gathered timings for the `ConstructCDS` and FOW prototypes. Table 4 lists the timings we collected: all times are user+sys times reported by the UNIX[14] `time` command. The first column of the table lists our subject programs, the second column gives the time that the `ConstructCDS` prototype required to construct the CDS for each program, and the third column gives the time required by the FOW prototype to construct the CDS for each program. All timings

[14]UNIX is a registered trademark licensed exclusively by Novell, Inc.

are arithmetic means of timings collected for five runs of the prototypes.

For subject programs with fewer than 90 source statements and 30 regions, the two prototypes exhibited similar performance. For example, the first row of Table 4 lists the times for `Bub.c`, a program with 38 source statements and 11 regions. Both prototypes required 0.7 seconds to compute the CDS for `Bub.c`. For larger programs, the efficiency of our `ConstructCDS` prototype becomes evident. For example, the second row of Table 4 lists the times for `Bub.long.c`, a program with 326 source statements and 92 regions. `ConstructCDS` required only 1.8 seconds to construct the CDS for `Bub.long.c`, whereas the FOW prototype required 16.7 seconds to construct that CDS. More dramatic savings are evidenced by the timings for `Find.long.c`, a program containing 579 source statements and 101 regions. For `Find.long.c`, the `ConstructCDS` prototype required only 1 second to construct the CDS, whereas the FOW prototype required 70.1 seconds.

Further experimentation with the two prototypes revealed that the bottleneck in the FOW prototype is the computation of postdominator information. To compute postdominator information for a function, the FOW prototype uses a straightforward algorithm that runs in time quadratic in the size of the function [4]. To illustrate the impact of postdominator computation on CDS construction time, column 4 of Table 4 records the time that the FOW prototype required to construct the CDS for each program excluding the cost of postdominator computation. A comparison of columns 3 and 4 of the table demonstrates the impact of the postdominator computation on the FOW prototype. For small programs, the time required to compute postdominator information affects the time required for CDS computation only marginally. For example, for `Bub.c`, postdominator computation accounts for only 0.1 second of the 0.7 seconds required to compute the CDS using the FOW prototype. However, for most larger programs, the time required to compute postdominator information dominates the overall cost of the FOW prototype. For example, for `Find.long.c`, postdominator calculation accounts for 62.0 of the 70.1 seconds required to compute the CDS.

The timings excluding dominator costs for the FOW prototype give us a lower bound on the time that the prototype would require if it employed a more efficient algorithm for calculating postdominance information [43]. However, even using this lower bound, the timings show that `ConstructCDS` is almost always as efficient as, and in most cases more efficient than, the FOW prototype. For example, for `Calc.long.c`, the `ConstructCDS` prototype required 1.5 seconds to construct the CDS; excluding postdominance construction, the FOW prototype required 5.9 seconds. Moreover, in practice, even a more efficient postdominance calculation algorithm would still add time to the cost of running the `FOW` prototype.

Study 2

Our second study investigated 3066 C functions from the GNU C compiler `gcc`[15] [41], UNIX[16] utilities, SPEC benchmarks,[17] and Mothra software testing system [11], to determine how often our algorithm constructs the CDS during the program's parse, without needing to adjust control dependencies. Figure 17 shows the percentages of types of functions we encountered: 58.1% of the functions contained no structured or unstructured transfers of control, 37.7% of the functions contained structured transfers of control but no *goto* statements, and 4.2% of the functions contained *goto* statements.

For 58.1% of the functions studied, `ConstructCDS` constructs the CDS during the parse without re-

[15]Copyright (C) 1987, 1989 Free Software Foundation, Inc, 675 Mass Avenue, Cambridge, MA 02139.
[16]UNIX is a registered trademark licensed exclusively by Novell, Inc.
[17]Copyright (C) 1989, 1990, 1991, 1992 by Standard Performance Evaluation Corporation.

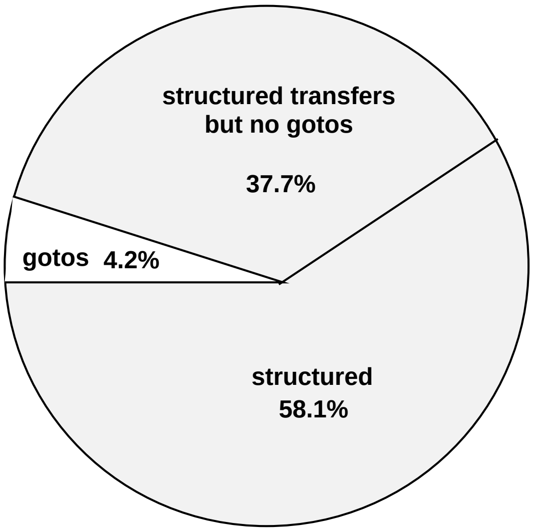


Figure 17: Relative percentages of procedures in experimental sample containing goto statements (4.2%), structured transfers of control only (37.7%), and no transfers of control (58.1%). The shaded portions of the graph show the percentage of procedures that `ConstructCDS` can handle without requiring explicit control flow information or additional graphs.

---

quiring computation of follow information. For the 37.7% of the functions that contain only structured transfers of control, `ConstructCDS` constructs the CDS during the parse without control flow or postdominator information, but with computation of follow information. For the remaining 4.2% of the procedures, `ConstructCDS` constructs the `CDS` during the parse, and attempts to recognize *goto* statements that are used as structured transfers of control. In cases where *goto*s cannot be recognized as structured transfers of control, `ConstructCDS` calls `Adjust()` to adjust control dependencies using control flow information. Thus, for 95.8% of the procedures analyzed, `ConstructCDS` constructs the CDS without control flow or postdominator information; this percentage is shaded in Figure 17.

5 Conclusion

We have presented an 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 control flow or postdominator information. For programs that contain unstructured transfers of control, an additional step adjusts the 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 implicitly. When necessary, the algorithm inserts edges representing control flow. Our approach has several advantages. Because we typically do not require control flow or postdominator information to construct a PDG, we can construct PDGs during the parse. Moreover, because our PDG contains control flow information, it supports applications that normally require control flow graphs, such as data flow analysis, test case generation, regression testing, and dynamic execution profiling.

Acknowledgements

Brian Malloy contributed to early work on this research. Robert Ballance and Barney Maccabe introduced the idea of a follow region for structured control transfers and the notion of "adjusting" the CDS in the presence of goto statements. Rama Tummala and Amar Yalavarthy implemented the `ConstructCDS` algorithm.

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] H. Agrawal, J. Horgan, E. Krauser and S. London, "Incremental regression testing," *Proceedings of the Conference on Software Maintenance '92,* pp. 348–357, September 1993.

[4] A. Aho, R. Sethi and J. Ullman, *Compilers, Principles, Techniques and Tools*, Additon-Wesley Publishing Company, 1986.

[5] T. Ball and J. Larus, "Optimally profiling and tracing programs," *Conference Record of the Nineteenth ACM Symposium on Principles of Programming Languages*, pp. 59–70, January 1992.

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

[7] S. Bates and S. Horwitz, "Incremental testing using program dependence graphs," *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages,* pp. 384–396, January 1993.

[8] 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.

[9] R. Cytron, J. Ferrante and V. Sarkar, "Compact representations for control dependence," *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, SIGPLAN Notices, vol. 25, no. 6, pp. 337–351, June 1990.

[10] R. Cytron, J. Ferrante, B. Rosen, M. Wegman and F. Zadeck, "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.

[11] R. DeMillo, D. Guindi, K. King, W. M. McCracken, and A. J. Offutt, "An extended overview of the Mothra software testing environment," *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pp. 142–151, July 1988.

[12] 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.

[13] 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.

[14] P. Frankl and E. Weyuker, "An applicable family of data flow testing criteria," *IEEE Transactions on Software Engineering,* vol. 14, no. 10, pp. 1483–1498, October 1988.

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

[16] 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.

[17] M. J. Harrold and M. L. Soffa, "An incremental approach to unit testing during maintenance," *Proceedings of the Conference on Software Maintenance '88,* pp. 362–367, October 1988.

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

[19] 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.

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

[21] M. J. Harrold and G. Rothermel, "A system for analysis and testing of C programs", Proceedings of Quality Week '93, May 1993.

[22] M. J. Harrold and B. A. Malloy, "A unified interprocedural program representation for a maintenance environment," *IEEE Transactions on Software Engineering*, pp. 584–593, June 1993.

[23] M. J. Harrold, "Performing data flow testing in parallel," Proceedings of the 8th International Parallel Processing Symposium (IPPS), pp. 200-207, April 1994.

[24] M. J. Harrold, L. Larsen, J. Lloyd, D. Nedved, M. Page, G. Rothermel,M. Singh, M. Smith, "Aristotle: A system for development of program analysis based tools," *Proceedings of 33rd Southeast Conference*, pp. 110–119, March 1995.

[25] J. Hartmann and D. Robson, "Techniques for selective revalidation," *IEEE Software*, pp. 31–36, January 1990.

[26] C. A. R. Hoare, "Algorithm 65: FIND," *Communications of the ACM*, vol. 4, no. 1, pp. 321–322, July 1961.

[27] C. A. R. Hoare, "Proof of a Program: FIND," *Communications of the ACM*, vol. 14, no. 1, pp. 39-45, January 1971.

[28] S. Horwitz, J. Prins and R. Reps, "Integrating non-interfering versions of programs," *ACM Transactions on Programming Languages and Systems*, vol. 11, no. 3, pp. 345–387, July 1989.

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

[30] B. Kernighan and D. Ritchie, The C Programming Language, Prentice-Hall, Englewood Cliffs, NJ, 1978.

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

[32] B. Korel, "Automated software test data generation," *IEEE Transactions on Software Engineering,* vol. 16, no. 8, pp. 870–879, August 1990.

[33] J. Laski and B. Korel, "A data flow oriented program testing strategy," *IEEE Transactions on Software Engineering,* vol. 9, no. 3, pp. 347–354, May 1983.

[34] S. Ntafos, "An evaluation of required element testing strategies," *Proceedings of the Seventh International Conference on Software Engineering*, pp. 250–256, March 1984.

[35] T. Ostrand and E. Weyuker, "Using data flow analysis for regression testing," *Proceedings of Sixth Annual Pacific Northwest Software Quality Conference,* pp. 58–71, September 1988.

[36] S. Rapps and E. Weyuker, "Selecting software test data using data flow information," *IEEE Transactions on Software Engineering*, vol. 11, no. 4, pp. 367–375, April 1985.

[37] G. Rothermel and M. J. Harrold "A safe, efficient algorithm for regression test selection," *Proceedings of the Conference on Software Maintenance '93,* pp. 358–367, September 1993.

[38] G. Rothermel and M.J. Harrold, "Selecting tests and identifying test coverage requirements for modified software", Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA 94), pp. 169–184, August 1994.

[39] B. G. Ryder and M. C. Paull, "Elimination algorithms for data flow analysis," *ACM Computing Surveys*, vol. 18, no. 3, pp. 277–316, September 1986.

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

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

[42] A. Taha, S. Thebut and S. Liu, "An approach to software fault localization and revalidation based on incremental data flow analysis," *Proceedings of COMPSAC 89,* pp. 527–534, September 1989.

[43] R. Tarjan, "Testing flow graph reducibility," *Journal of Computer and System Sciences*, vol. 9, pp. 355–365, 1974.

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

A Detailed Version of ConstructingCDS

algorithm `ConstructCDS`
input `AST` : abstract syntax tree for procedure `P` with root *pgm*
output `CDS` : control dependence subgraph for `P`, with control flow information

declare
types:

`ASTNodeType`: (* `AST` node information *)
`type`: type of `AST` node
`info`: other information, such as uses and defs

`NodeType` : ('statement', 'predicate', 'exit', 'entry', 'if-clause', 'else-clause', 'while header', 'while body', 'while exit', 'summary', 'label')

`CDSNodeType`: (* `CDS` node information *)
`type` : `NodeType`
`cfpredlist`: pointer to list of immediate implicit or explicit control flow predecessors
`cfsucclist`: pointer to list of immediate implicit or explicit control flow successors
`cdpredlist`: pointer to list of immediate control dependence predecessors
`cdsucclist`: pointer to list of immediate control dependence successors

`CDStackEntryType` (* `CDStack` entries *)
`node`: pointer to a `CDSNodeType`
`cfunreslist`: list of nodes in `CDS` needing control flow edge resolution

`LoopStackEntryType`: (* `LoopStack` entries *)
`node`: pointer to `CDSNodeType`
`breaklist`: nodes representing break statements needing control flow edges resolved
`possiblebreaks`: list of pairs (Pn,V) where Pn is a predicate node and V = 'T' or 'F'

`RegionTableEntryType` (* `RegionTable` entries *)
`node` : pointer to `CDSNodeType`
`cdepconditions` : predicate paths associated with the node

`TransferListEntryType` (* `TransferList` entries *)
`type` : pointer to `NodeType`
`continue`: list of predicate paths
`break`: list of predicate paths
`return-or-exit`: list of predicate paths

`LabelTableEntryType`: (* `LabelTable` entries *)
`label`: name of label
`region`: region node created for label
`gotolist`: list of statement nodes that need control flow edges to the label region node

variables:
`ASTNode`, `LookAhead`: `ASTNodeType` (* information on `AST` nodes *)
`CDS`: `CDSNodeType` (* pointer to the entry node in `CDS` – initially null *)
`CDStack`: stack of pointers to `CDStackEntryType` (* region/predicate nesting – initially empty *)
`Active`: pointer to `CDStackEntryType` (*region on top of `CDStack` *)
`LoopStack`: stack of pointers to `LoopStackEntryType` (* loop-specific information *)
`RegionTable`: array of `RegionTableEntryType` (* regions and their control dependencies – initially empty *)
`TransferList`: array of `TransferListEntryType` (* control transfer information – initially empty *)
`LabelTable`: array of `LabelTableEntryType` (* information about labels – initially empty *)
`MostRecentNode`: pointer to `CDSNodeType` (* most recently created `CDS` node *)
`BreakNode, ExitNode, IfClauseNode, IfElsePredicateNode, LabelNode, ParentPredicate, PredicateNode, StatementNode, SummaryNode, WhileBodyNode, WhileExitNode, WhileHeaderNode, UnResolvedNode` (* `CDS`, `CDStack` temps *)
`AdjustFlag`: boolean (* set if a procedure has unstructured gotos *)
`Label`: labelstring (* holds a 'label' (goto target) *)
`Found`: boolean (* used to search `LabelTable` *)
`Location`: integer (* used to search `LabelTable` *)

procedure AddNode(T:NodeType,N:CDSNodeType,L:EdgeLabel): pointer to CDSNodeType
(* creates CDS node with type T using MakeCDSNode, adds it to CDS under N with edge label L using AddtoCDS;
if node is a region node, adds it to RegionTable with its associated predicate path information; returns node *)

procedure MakeCDSNode(T:NodeType): CDSNodeType
(* creates CDS node of type T, adds edges from unresolved nodes if necessary *)

procedure AddtoCDS(P,N:CDSNodeType, L:EdgeLabel)
(* adds N to CDS under P, with edge label L if L is non-null; *)

procedure Push(N:CDSNodeType)
(* creates new record for N, initializes it, and pushes it onto CDStack *)

procedure Pop(): pointer to CDStackEntryType
(* pops CDStack, passing .cfunreslist down to next entry in CDStack if necessary *)

procedure GetASTNode(): ASTNodeType
(* returns information about the next node in the AST, as could be determined in a preorder traversal of the AST *)

procedure PushLS(N:CDSNodeType)
(* makes new LoopStack entry for N *)

procedure PopLS()
(* pops LoopStack, resolves loop-related edges *)

procedure Search(L:Label, Found:boolean, Index:integer)
(* finds L, or a location where L should go, in LabelTable *)

procedure ComputeFollow(N:CDSNodeType)
(* given a predicate node in the CDS, computes the follow region, updates CDStack to reflect the new active region;
uses Follow, a list of regions, NewActive, a CDSNodeType, TempCond, a disjunction of path conditions, and
TempList, a list of CDS nodes that need control flow edge resolution *)

```
begin
    (* First task: compute the follow region for statements following control structure controlled by N *)
    if N is an 'if-else' predicate node then
        TempCond := the control dependence conjunction of the control dependence negations of the relevant portions
            of the Pis in TransferList[N].continue, TransferList[N].break, or TransferList[N].return-or-exit
    else (* N is a 'while' predicate node *)
        TempCond := the control dependence conjunction of the control dependence negations of the relevant portions
            of the Πs in N.return-or-exit disjoined with the control dependence disjunction of the relevant
            portions of the Πs in N.break
    endif
    foreach P in TempCond do add region corresponding to P to Follow
    if Follow has one element then NewActive := Follow
    else (*multiple-node follow, create summary region *)
        NewActive := MakeCDSNode('summary')
        foreach entry R in Follow do MakeCDEdge(R, NewActive, -); Add NewActive to Active.cfunreslist endfor
    endif
    TempList := CDStack[top].cfunreslist; CDStack[top].cfunreslist := empty
    Pop() (* pop current active node *)
    Push(NewActive) (* push new active node on CDStack *)
    CDStack[top].cfunreslist := TempList
    (* Second task: pass lists of control transfers to nearest enclosing control structure, if necessary *)
    ParentPredicate := topmost predicate in CDStack, or 'entry'
    if ParentPredicate = if-else 'predicate' node then
        foreach Π in TransferList[N].continue, TransferList[N].break, or TransferList[N].return-or-exit do
            Add Π to the corresponding list in TransferList[ParentPredicate]
        endfor
    else if ParentPredicate = while 'predicate' node then
        foreach Π in TransferList[N].return-or-exit do
            Add Π to TransferList[ParentPredicate].return-or-exit
        endfor
    endif
end ComputeFollow
```

```
begin (* main *)
    while ( (ASTNode := GetASTNode()) ≠ end pgm ) do (* main program loop: process AST until done *)

      case ASTNode.type is

        pgm:
            CDS := MakeCDSNode('entry') (* create 'entry' region node as root of CDS *)
            Push(CDS) (* push 'entry' region node on CDStack *)
            ExitNode := MakeCDSNode('exit') (* make 'exit' node for later use *)

        statement:
            StatementNode := AddNode('statement', Active,-) (* add 'statement' node to CDS *)

        if-else:
            ASTNode := GetASTNode (* read condition node *)
            PredicateNode := AddNode('predicate', Active, -) (* add 'predicate' node to CDS *)
            Push(PredicateNode) (* push 'predicate' node on CDStack *)

        if-clause/else-clause: (* shown for if-clause; else-clause is similar *)
            IfClauseNode := AddNode('if-clause', Active, 'T') (* add 'if-clause' node to CDS *)
            Push(IfClauseNode) (* push 'if-clause' region node on CDStack *)

        end if-clause/end else-clause: (* same actions for end of both if-clause and else-clause *)
            Add MostRecentNode to Active.cfunreslist if it is not a transfer node
            while Active.type = 'label' do Pop() endwhile (* get rid of 'label' regions *)
            Pop() (* pop CDStack *)

        end if-else:
            IfElsePredicateNode := Pop() (* finished processing this if-else *)
            if any of TransferList[IfElsePredicateNode].{continue, break, return-or-exit} are not NULL
                then ComputeFollow(IfElsePredicateNode) (* unstructured transfers of control present *)

        while:
            WhileHeaderNode := AddNode('while header', Active, -) (* add 'while header' region node to CDS *)
            Push(WhileHeaderNode) (* push 'while header' region node onto CDStack *)
            PushLS(WhileHeaderNode) (* push 'while header' region node onto LoopStack *)
            ASTNode := GetASTNode (* read condition AST node *)
            PredicateNode := AddNode('predicate', Active, -) (* add 'predicate' node to CDS *)
            Push(PredicateNode) (* push 'predicate' node onto CDStack *)
            WhileBodyNode := AddNode('while body', Active, 'T') (* add 'while body' region node to CDS *)
            Push(WhileBodyNode) (* push 'while body' region node onto CDStack *)

        end while:
            while Active.type = 'label' do Pop() endwhile (* pop 'label' regions from CDStack *)
            Pop() (* pop Active region node from CDStack *)
            Add MostRecentNode to Active.cfunreslist if it is not a transfer node
            Resolve control flow edges for nodes on Active.cfunreslist and reinitialize Active.cfunreslist
            PredicateNode := Pop()
            WhileExitNode := AddNode('while exit', PredicateNode, 'F') (* add 'while exit' region node to CDS *)
            Add WhileExitNode to Active.cfunreslist
            foreach BreakNode on LoopStack[top].breaklist do MakeCFEdge(BreakNode,WhileExitNode) endfor
            if LookAhead.type is 'label' that matches target of LoopStack[top].possiblebreaks then
                foreach predicate pair in LoopStack[top].possiblebreaks matching label do
                    Add predicate pair to TransferList[predicate associated with LoopStack[top]].break
                    Remove it from LoopStack[top].possiblebreaks
                endfor
            endif
            TempCond := the control dependence conjunction of the control dependence negations of the relevant portions
                of the PPis in TransferList[N].break or TransferList[N].return-or-exit
            foreach P in TempCond do make a CD edge from region corresponding to P to loop header region
            WhileHeaderNode := Pop()
            PopLS()
            if TransferList[PredicateNode].return-or-exit is not empty then ComputeFollow(PredicateNode)
            if LoopStack[top].possiblebreaks is nonempty then (* gotos not breaks in this loop *)
                if another loop on LoopStack then pass .possible breaks to next loop
                else AdjustFlag = true (* goto is not break *)
                endif
            endif
```

```
        continue:
            StatementNode := AddNode(ASTNode.type, Active, -)
            if the topmost predicate node in CDStack is an if-else 'predicate' node then
                Add predicate path associated with Active to TransferList[topmost-predicate-on-CDStack].continue
            endif
            MakeCFEdge(StatementNode,LoopStack[top].node)

        break:
            StatementNode := AddNode(ASTNode.type, Active, -)
                Add predicate path associated with Active to TransferList[ topmost-predicate-on-CDStack].break
            Add StatementNode to LoopStack[top].breaklist

        return/exit:
            StatementNode := AddNode(ASTNode.type, Active, -)
                Add predicate path associated with Active to TransferList[topmost-predicate-on-CDStack].return-or-exit
            MakeCFEdge(StatementNode, ExitNode)

        goto:
            StatementNode := AddNode('goto', Active, -) (* add a goto node to the CDS *)
            Label := the label used by the goto (* make LabelTable entry *)
            Search(LabelTable,Label,Found,Location)
            if (Found) and LabelTable[location].region is not empty then
                if leftmost child of LabelTable[Location].region is 'while header' on top of LoopStack then
                    (* goto is continue *)
                    if the topmost predicate node in CDStack is an if-else 'predicate' node then
                        Add predicate path associated with Active to TransferList[topmost-predicate-on-CDStack].continue
                    endif
                else AdjustFlag := true (* backwards branch is not a continue *)
                endif
                MakeCFEdge(StatementNode, LableTable[Location].region)(* create control flow edges from goto *)
            else (* assume goto is break *)
                LabelTable[Location].label := Label (* add this label to LabelTable *)
                (* save goto location for resolution when label is found *)
                append StatementNode to LabelTable[Location].gotolist
                Add predicate path associated with Active to LoopStack[top].possiblebreaks
                Add predicate path associated with Active to TransferList[topmost-predicate-on-CDStack].breaks
            endif

        label:
            LabelNode := AddNode('label', Active, -) (* add label region to CDS *)
            Push(LabelNode) (* push 'label' region node on CDStack *)
            Search(LabelTable,Label,Found,Location) (* find location for this new label *)
            if (not Found) then (* first time this label is referenced *)
                LabelTable[Location].label  := Label (* add label node to LabelTable *)
                LabelTable[Location].region := LabelNode (* record region associated with label *)
            else (* label has been previously referenced *)
                foreach StatementNode on LabelTable[Location].gotolist do
                    MakeCFEdge(StatementNode, LabelTable[Location].region) (* control flow edge from goto *)
                endfor
            endif

      endcase

    endwhile

(* end pgm encountered, loop exited *)
    while Active.type = 'label' do Pop() endwhile (* pop 'label' region nodes *)
    AddtoCDS('entry', ExitNode) (* connect the 'exit' node into the CDS *)
    if MostRecentNode is not a transfer node then MakeCFEdge(MostRecentNode,ExitNode) endif
    if Active.cfunreslist is not empty then (* resolve any other unresolved control flow edges *)
        foreach node UnresolvedNode on Active.cfunreslist do MakeCFEdge(UnresolvedNode,ExitNode) endfor
    endif
    if AdjustFlag then Adjust() endif
    remove spurious region nodes
    return(CDS)
end (* main *).
```