Program Dependence Graphs for the Rest of Us

Robert A. Ballance
Arthur B. Maccabe
Department of Computer Science
The University of New Mexico
Albuquerque, NM 87131
Technical Report 92-10

August 1992
Revised, October 1992

This work was supported in part by NSF under grant CCR-9110954, by Sandia National Laboratories under contract No. AC-6748, and by Kachina Technologies, Inc.

Abstract

This report presents new control dependence analysis techniques that succeed in constructing a control dependence graph (CDG) in all of the common cases without requiring either the control flow graph or the auxiliary structures needed by the fully general algorithm. In the worst case, the intermediate structures built by our algorithms are used to derive a simplified form of the control flow graph that is then used by the general algorithm. In this eventuality, the general algorithm may run more quickly, since a portion of its analysis has already been performed.

The report also presents an adaptation of Tarjan's interval analysis algorithm for data flow analysis that uses the control dependence graph instead of the control flow graph. Using the CDG-based analysis algorithm allows us to construct a program dependence graph (PDG) without first constructing a control flow graph.

This approach offers several advantages over the conventional models. It eliminates a complete program representation (the control flow graph), it simplifies the construction of the PDG and it supports efficient data flow analysis using the internal representations available. Also, understanding the direct construction of a control dependence graph increases one's understanding of control dependence and its relation to control flow.

Finally, this report presents a case study of `goto` usage in C programs which confirms the assumptions that (1) `goto`s are relatively infrequent and (2) `goto` usage in C programs largely follows predictable patterns that can be exploited to produce PDGs without resorting to global control dependence analysis in many cases.

Keywords: Program dependence graph, control dependence, data flow analysis, elimination algorithms, `goto` usage, C programming language.

1.0 Introduction

Program representations that combine control dependence and data dependence information are receiving widespread attention by researchers and implementors of language based tools. Often, the program representation being used is some variant of a program dependence graph (PDG) [5], since it encodes control dependence and data dependence information uniformly.

Although the PDG unifies control dependence and data dependence, techniques for constructing a PDG rely on the presence of a control flow graph [1]. During construction of the PDG, the control flow graph is used to identify the control dependencies and data dependencies. Direct construction of the control dependence graph (CDG) (the control dependence portion of a PDG) for programs containing only conditionals, while loops, and simple statements has long been understood. This report extends that construction to programs using multiple level break and continue constructs, function exits, and well behaved `goto`s. We then show how to derive a variant of Tarjan's interval analysis algorithm [11] for data flow analysis which operates on the control dependence, rather than the control flow, graph. Thus, for well behaved procedures, the PDG can be obtained without constructing the control flow graph.

The techniques presented here succeed in constructing the PDG in all of the common cases without requiring either the control flow graph or the auxiliary structures needed by a fully general algorithm. For virtually all procedures, the control dependence graph is completely constructed without using the control flow graph. In the worst case, the intermediate structures built by our algorithms are used to derive a simplified form of the control flow graph that is then used by the general algorithm. In this eventuality, the general algorithm may run faster, since a portion of its analysis has already been performed. The CDG is extended into a PDG by using the results of the data flow analysis to add data dependence edges.

This approach offers several advantages over conventional models.

- One complete program representation (the control flow graph) is eliminated.
- Construction of the PDG is simplified.
- Efficient data flow analysis using the CDG becomes available.
- Understanding the direct construction of the control dependence graph increases one's understanding of control dependence and its relation to control flow.

Given the number of PDG-based representations and the number of algorithms that use such representations, PDG construction will become a common technique and will have many applications.

2.0 Definitions

A program dependence graph encodes both control and data dependence information using directed edges connecting graph vertices. The vertices in a PDG may represent several concepts: data operators, branch points in control flow, or regions of common control dependence. Edges represent any of several varieties of data dependence or control dependence. Auxiliary edges may be present to indicate special kinds of data flow,

such as different kinds of procedure parameter passing. For practical purposes, every PDG is rooted at a distinct vertex named Entry. We assume every vertex in a PDG is reachable from Entry via some combination of data or control dependence edges.

In this report, we assume that the reader is familiar with program dependence graphs [5] and control dependence, the usual algorithms for performing control dependence analysis [3], and the usual algorithms for data flow analysis [1] including Tarjan's interval analysis [8] [9] [11].

2.1 Preliminary Definitions

One can think of a PDG as a layering of two different subgraphs: a data dependence subgraph (DDG) and a control dependence subgraph (CDG). The data dependence subgraph consists of all edges that indicate data dependencies together will all of the vertices that are either sources or targets of data dependence edges.

Definition 1 Let $P = \langle V, E \rangle$ be a program dependence graph. The *data dependence subgraph* of P is the subgraph $DDG_P = \langle V', E' \rangle$ where $E' = \{e \in E : e \text{ denotes a data dependence}\}$ and $V' = \{v, v' \in V : \langle v, v' \rangle \in E'\}$.

The control dependence subgraph includes all of the edges that indicate control dependence, together with the vertices that are the sources (not the targets) of such edges. The root vertex Entry is also included in the CDG.

Definition 2 Let $P = \langle V, E \rangle$ be a program dependence graph. The *control dependence subgraph* of P is the subgraph $CDG_P = \langle V', E' \rangle$ where $E' = \{e \in E : e \text{denotes a control dependence}\}$ and $V' = \{v \in V : \langle v, v' \rangle \in E'\} \cup \{\text{Entry}\}$

2.2 Control Dependence

Control dependence formalizes the notion that execution of one vertex in the graph conditionally depends upon the execution of another vertex in the graph. Thus, in a linear sequence of statements, none of the statements will control depend upon any of the others. In the absence of data dependencies, two vertices that have identical control dependencies could be executed simultaneously. Control dependence is formally defined in terms of the postdominance relationship on the control flow graph.

Definition 3 A *control flow graph* (CFG) is a directed graph whose vertices include the basic blocks of a program along with two distinguished vertices Entry and Exit. There is an edge from Entry to any basic block at which the program can be entered, and there is an edge to Exit from any block that can exit the program. To ensure that the control dependence graph is rooted at Entry, there is also an edge from Entry to Exit.

Definition 4 Let X and Y be vertices in the CFG. If Y appears on every path from X to Exit, then Y *postdominates* X. If Y postdominates X, and $X \neq Y$, then Y *strictly post–dominates* X. The *immediate postdominator* for X is the closest strict postdominator of X. A graph of the immediate postdominator relationships is called a *postdominator tree*. By construction, the postdominator tree is rooted by Exit.

The notion of postdominance is crucial to defining control dependence. Suppose that X and Y are vertices in the control flow graph, and that Y postdominates X. Then every path from X to Exit passes through Y. Thus, whenever X is executed, Y will eventually be executed. In this case, there is no *conditional* control dependence between X and Y; knowing that X will execute tells us that Y will also execute.

Conversely, suppose that Y does not postdominate X. If there is no path in the control flow graph between X and Y, then the relationship between X and Y is arbitrary, and neither's execution can be said to depend upon the other. If there is at least one path from X to Y in the control flow graph, execution of Y conditionally depends on the execution of X. It is only in this case that Y is control dependent upon X. Definition 5 formalizes this notion, with the additional constraint that X be the "closest" branch point in execution order.

Definition 5 A CFG vertex Y is *immediately control dependent* on a CFG vertex X if both of the following hold:

1. There is a nonnull path $p: X \Rightarrow^{+} Y$ such that Y postdominates every vertex after X on p.
2. The vertex Y does not strictly postdominate the vertex X.

Definition 6 A CFG vertex Y is *control dependent* on a CFG vertex X if either Y is immediately control dependent upon X, or there is a vertex Z such that Y is immediately control dependent upon Z, and Z is control dependent upon X.

2.3 Partitioned Program Dependence Graphs

The definition of control dependence can be applied to construct a PDG directly. However, for many purposes, it is useful to partition the vertices of the PDG into collections sharing the same control conditions. The partitions can be directly encoded in the PDG by adding region vertices.

Definition 7 A *region* is a new kind of vertex, present only in the control dependence subgraph, that represents a set of control conditions. Each child of a region is control dependent on every parent of a region. All of the edges incident to a region vertex indicate control dependence. The special Entry vertex is, by definition, a region having no control predecessors.

Definition 8 A CDG is *partitioned* if region vertices have been inserted such that every operator appearing in the original program is the child of some region while every region is either the child of a predicate or the child of some other region.

Partitioning the CDG introduces a new class of control dependence edges as well as a new kind of vertex. Edges from predicates to region vertices indicate immediate control dependence. However, edges emanating from regions indicate transmission, or flow, of control dependence conditions from the ancestors of the region to the descendants of the region. Since the region vertex encapsulates a set of control conditions, each child of a region vertex is control dependent upon all of the predecessors of that region. Thus, while control dependence edges from a predicate to a region denote actual control

dependencies, edges that emanate from regions only transmit the dependencies from the in edges of the source region to the out-edges of the source region.

Figure 1 shows the partitioning that results from factoring the control dependencies

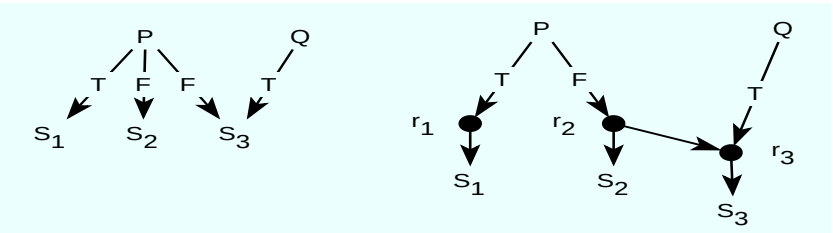


FIGURE 1. A flat CDG and its partitioned counterpart

from predicates P and Q into region vertices. Region vertices, represented by solid circles have been introduced in the partitioned graph. Note that S_3 in region r_3 is control dependent on either P being false or Q being true.

Regions act like an extended basic block in a control flow graph. Cytron, Ferrante, and Sarkar [3] note that the partitioned CDG may require quadratic space to represent all of the regions; their paper discusses more compact encodings.

Most of the work published on PDGs does not try to order the children of a region. However, as we shall show, by ordering the children of a region according to their appearance in the program, one can rederive the original sequential representation. This allows certain CFG-based algorithms, such as data flow analysis, to be easily adapted to a CDG-based representation [7]. Ultimately, of course, only those orderings forced by data or control dependencies are relevant.

Definition 9 A PDG is *ordered* if there is a traversal of the vertices within a region that visits the vertices in their original order.

Redundant regions can be introduced in the construction of the partitioned graph, or they can arise during subsequent processing. By a minimal CDG, we mean one which contains no unnecessary regions.

2.4 Reading a CDG

Control dependence graphs can be quite difficult to read unless the following guidelines are remembered:

- A control dependence edge from a predicate P to a region R indicates that all of the children of R are control dependent upon P. Edges that emanate from predicates are usually labeled with the applicable control condition (true or false).
- An edge that connects a region R to a predicate P indicates that the execution of P is control dependent upon the predecessors of R. Such an edge is interpreted just like an edge from any region to any data producing operator in the graph.

- An edge from a region R_1 to a region R_2 indicates that the control conditions under which the successors of R_2 can execute are a subset of the conditions under which the successors of R_1 can execute. Edges from regions to regions are normally introduced by explicit transfers of control.
- An edge that connects two predicates would be an error. In a properly partitioned PDG, every control dependence successor of a predicate must be a region vertex.

3.0 Construction of the Control Dependence Graph

The construction of a PDG involves two steps: control dependence analysis to uncover control dependencies and data flow analysis to uncover data dependencies. Conventionally, both kinds of analysis are based upon a control flow graph—a directed graph in which the vertices are linearly executed sequences of data manipulations and the edges represent possible flow of control between blocks.

For simple if-while programs, it has long been known that control dependence information can be directly derived from the control structures in the program. However, for more complex or even unstructured programs, control dependence analysis has been driven by the control flow graph of the program being analyzed. General algorithms for computing control dependence information construct both the reverse of the control flow graph and its dominator tree[1]. The dominator tree encodes the global program flow information needed to compute a control dependence graph (CDG). Once the dominator tree has been computed, the technique of dominance frontiers [3] can be used to compute control dependencies. Following that, one can partition the resulting CDG by adding region vertices.

General control dependence analysis algorithms [3] [5] deal with programs containing arbitrary `goto`s. In that case, global analysis of the program using the control flow graph is necessary. However, in many simpler cases, one can derive the CDG directly from the control constructs present in the text of a program. In this section, we present three variants of CDG-producing algorithms which operate on the abstract syntax tree representation of a program. The first two actually produce an ordered, partitioned CDG. The third, which copes with arbitrary `goto`s, produces a simplified CFG that can then be used with a more general control dependence analysis algorithm.

3.1 CDGs for Structured Programs

Consider programs composed from the grammar shown in Figure 2. In the absence of explicit transfer of control, construction of the CDG is straightforward. Each predicate in the program introduces one or two new regions into the CDG. Graphically, we can represent the transformations as in Figure 3). The algorithm to perform the transformations is a simple preorder tree walk over the syntax tree. During the tree walk, a stack of

1. The reverse of the control flow graph is just the control flow graph with each edge's direction reversed. It is always the case that the dominator tree of the reversed control flow graph will be the postdominator tree for the original graph.

$$
\begin{aligned}
S \quad & : \textit{simple-stmt} \\
& \mid \text{if } P \text{ then } \textit{stmts} \\
& \mid \text{if } P \text{ then } \textit{stmts}_1 \text{ else } \textit{stmts}_2 \\
& \mid \text{while } P \text{ do } \textit{stmts} \\
& \mid \text{do } \textit{stmts} \text{ while } P
\end{aligned}
$$

FIGURE 2. While Language L_0

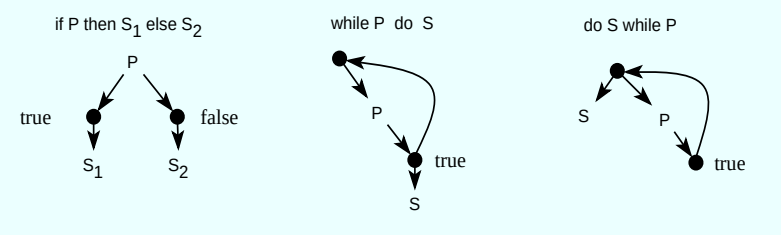


FIGURE 3. Abstract Syntax to CDG transforms for language L_0

region vertices is manipulated; the top region in the stack will be the "current region" to which most new vertices are added. Initially, the region stack contains the regions representing Entry. The basic steps are:

Simple Statements: Make statement control dependent upon the current region.

Conditionals: Create a predicate vertex and two new regions—one for the true portion of the conditional, and one for the false portion. Make both new regions control dependent upon the predicate.

1. Make the predicate controlling the conditional control dependent upon the current region.
2. Push the region representing the true control dependence onto the stack, visit the true part, and then pop the stack.
3. If there is a false part, push the region representing the false control dependence onto the stack, visit the false part, and then pop the stack.

While-loops: Create two new regions—one for the body of the loop and one for the header of the loop. The header region will be the target of the back edge, while its first child will be the predicate that controls the loop.

1. Make the header region control dependent upon the region atop the stack. Push the header region onto the region stack.
2. Make the predicate control dependent upon the header region. Make the body region control dependent upon the predicate.
3. Push the body region onto the stack and visit the body.
4. Make the header region control dependent upon the body region (thus creating the back edge).
5. Finally, pop both the header region and the body region from the stack.

Do-while loops: Treat like while loops, except that the body of the loop is added to the header region.

```
S   : simple-stmt
    | if P then stmts1
    | if P then stmts1 else stmts2
    | while P do stmts
    | do stmts while P
    | break
    | continue
    | return E
```

FIGURE 4. Language L_1

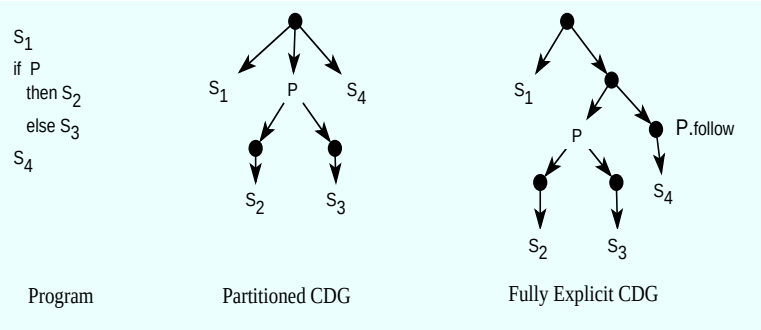


FIGURE 5. Example of Follow Regions

3.2 Structured Exits

Structured exits, like C's break, continue, and return, can be added to the simple algorithms with only a few modifications. However, the following theoretical framework is needed to justify the results. Basically, the theorems establish a simple property: when a block is exited, a new region must be created that will become the "current region" for statements following the block. In the following, assume that programs are written in the subset of C defined by the grammar of Figure 4.

It is valid (but not usual) to explicitly introduce a new region for every statement. This is just a representation of control dependence in which every implicit transfer of control is explicitly shown. Figure 5 shows a simple program with all possible region vertices introduced. The region tagged "P.follow" summarizes the control dependence information for the statements following the conditional statement P. Addition of explicit follow regions is similar to using the standard binary tree encoding for linear lists.

Definition 10 Let R be a region with children $S_1, \ldots, S_n$. It is always valid to replace R with two regions R' and S_k.follow such that

1. R' has children $S_1, \ldots, S_k$.
2. S_k.follow has children $S_{k+1}, \ldots, S_n$.
3. There is a edge from R' to S_k.follow.
4. R' is the direct replacement for R.

When dealing with structured programs, regions like P.follow are unnecessary since its only control dependence predecessor must be the region that contains P. However, when structured jumps are present, regions like P.follow become necessary.

In the absence of transfers of control, each statement at a single nesting level is in the same region of control. When control transfers are introduced, this invariant is violated. For instance, consider the two flow graphs shown in Figure 6. The left-hand graph is a conditional statement with a single exit. In this case, S_3 is not control dependent upon P. However in the right-hand graph, S_3 is control dependent upon P, since the transfer of control following S_1 skips over S_3.

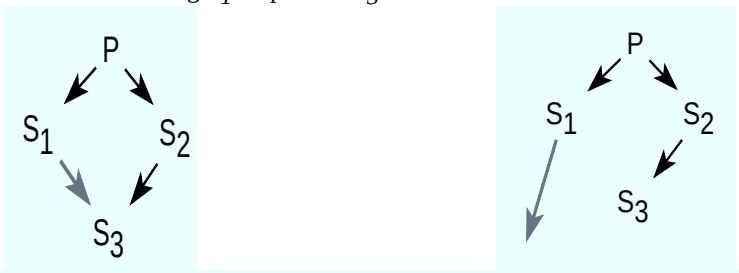


FIGURE 6. How transfer of control affects conditional constructs

Lemma 1 Let S be a complex statement (e.g., conditional, loop) with P as its controlling predicate. Suppose that S has nested within it an exit (e.g. break, continue, return) that is immediately control dependent on P when the value of P is L. Let S.follow be the region that summarizes the control conditions that hold after S is executed. Suppose that the target of the exit postdominates S.follow. If S_1 is the first statement in S.follow, then S_1 is control dependent on P when the value of P is $\neg L$.

Proof: If S did not contain any embedded exits, then S_1 would postdominate all of S. In particular, S_1 would postdominate P. However, since S contains an exit that forces execution to skip over S_1 when the value of P is L, S_1 no longer postdominates P. But if the value of P is not L, S_1 continues to postdominate any statement control dependent on P. Hence S_1 is control dependent on P when the condition $\neg L$ holds.

Definition 11 Given a sequence of predicates $\Pi = \{P_1, \ldots, P_n\}$, define the value of Π to be the value of the boolean expression $E(\Pi) = P_1 \wedge P_2 \wedge \ldots \wedge P_n$. Define the *control dependence negation* of Π, denoted $\neg\Pi$, to be the disjunction

$$
\begin{aligned}
& P_1 \wedge P_2 \wedge \ldots P_{n-1} \wedge \neg P_n \\
\vee\ & P_1 \wedge P_2 \wedge \ldots \wedge \neg P_{n-1} \\
& \cdots \\
\vee\ & \neg P_1 \\
\equiv\ & \neg P_1 \vee \neg P_2 \vee \ldots \vee \neg P_n
\end{aligned}
$$

Theorem 1 Let S be a complex statement with P as its controlling predicate. Suppose that S has nested within it a transfer of control that is controlled by the predicate path $\Pi = \{P_1, P_2, \ldots, P_{n-1}\}$, $n \geq 1$, (relative to P in the abstract syntax tree), and that the target of the exit postdominates S.follow. If S_1 is the first statement in S.follow, then S_1 is control dependent upon $\neg\Pi$.

Proof: By induction on the length of Π.

Consider $\Pi = \{P\}$. In this case, the exit is directly within S, so Lemma 1 applies. Now suppose that the theorem holds for all predicate paths Π of length *n*, and that the

exit in S is controlled by the predicate path $P \cdot \Pi$. Then S_1 is executed if either $\neg P$ holds or if the exit was not taken, that is, whenever $\neg P \vee (P \wedge \neg \Pi)$ holds true. But this is just $\neg (P \cdot \Pi)$.

Theorem 2 Let S be a break (return, exit) statement nested within *n* compound statements. The program point that is the target of *S* postdominates all statements occurring between *S* and its target in a preorder traversal of the abstract syntax tree.

Proof: This is a direct consequence of structured exits. Preorder traversal, in this case, simulates possible execution order.

Theorem 3 Let *S* be a continue statement nested within *n* compound statements. The program point that is the target of *S* postdominates all statements occurring between (in depth-first order) *S* and the end of the compound statement that forms the body of the loop which contains *S.*

Proof: Follows directly from the definition of continue.

Using Theorems 1–3, one can derive a new algorithm for generating a CDG from a parse tree (or abstract syntax tree) based on the language L_1. Such an algorithm is summarized below. In the construction, for each compound statement *S*, the region *S*.follow is created. Like the structured (L_0) case, when a new complex structure is visited, we create several regions. In addition to the region stack maintained in the simple case, a stack summarizing control predicates is maintained. Within this second stack, called the "condition stack," are entries of the form <construct-type, Pred, followRegion, loopHdr> where construct-type denotes the type of structured context (conditional, loop, etc.), Pred is the region controlled by the negation of the predicate condition being traversed, followRegion is the region that follows the construct, and loopHdr is the region that will be the target of a continue. If the current construct is not a loop, loopHdr is not defined.

Figure 7 shows a snapshot of the algorithm as the program in the figure is being analyzed. The snapshot was taken just after the break statement was processed. Black edges have already been added to the graph; gray edges remain to be added. The tree walk is in Q's true region, and the condition stack has been traversed to make Q.follow depend on the region Q.false. However, Q.false has not yet been visited. (It will be, once Q's true part has finished.) Two constructs are on the construct stack, one for the loop at P, and

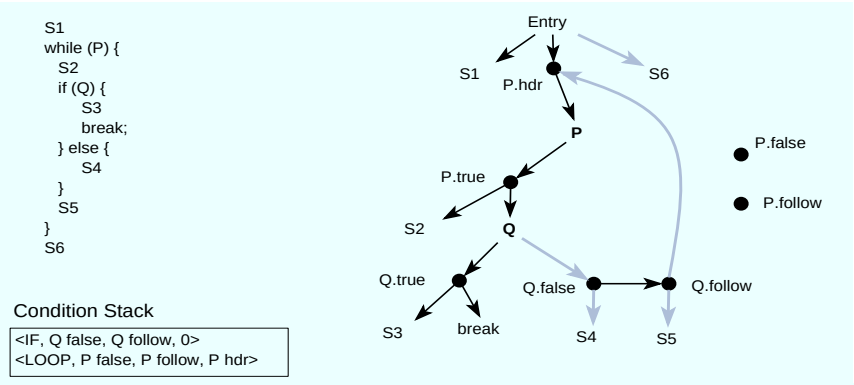


FIGURE 7. Construction of a CDG with structured exits

one for the conditional at Q. No edges involving P.false or P.follow will be needed in the final graph. A complete description of the algorithm appears in Section 8.0 .

3.3 Handling arbitrary `goto`s

The algorithm given above cannot, in general, handle arbitrary `goto`s. However, within our approach, two alternatives are possible. Many uses of `goto`s in C programs are emulations of multi-level break, multi-level continue, or procedure exits. When such uses can be easily discerned, the above algorithm can be adjusted to handle them. Otherwise, our algorithm can produce a partitioned variant of the CDG, called the "not-quite" CDG (NQCDG) that contains all possible regions of control, but does not contain the correct control dependence edges. From the NQCDG, one can derive a variant of the reverse control flow graph in which every vertex is a region that will appear in the CDG. This "reduced" CFG can therefore serve as input to CFG-based control dependence analysis algorithms.

Extending the algorithm to arbitrary `goto`s requires the following observation:

Theorem 4 Let S be a compound statement containing a statement with label L, let G be a `goto` from a region R (not nested inside S) to L. Then the statement labelled by L shares the control dependencies of R.

Proof: Edges between connecting regions transmit control dependencies, but do not add new ones. Any time the region containing the `goto` is executable, the region containing the target must also be executable.

To extend the conversion algorithm to handle arbitrary `goto`s, add the following steps:

1. Each `goto` introduces an edge from its current region to the region controlling its target.
2. Each `goto` introduces a new region following the `goto`. (This step is only needed if dead code may be present.)
3. Each label begins a new region.
4. After each complex statement S, add S.follow to the CDG.

These four steps assure that the necessary regions are created, but will not assure that the correct control dependencies will be present.

One can convert the resulting graph into a reduced, reverse CFG in a single preorder traversal. During the preorder traversal, one links regions in the NQCDG to their nominal (sequential) predecessors in the CFG. Figure 8 illustrates the before and after situation. The left-hand graph in Figure 8 is the NQCDG for the program shown in Figure 7. The reverse links to be established are shown as dotted lines in the graph on the right-hand side of the figure. Regions are shown using subscripted labels "r_i" rather than simple dots, in order to improve readability. A preorder traversal of the NQCDG suffices, in

which predecessor regions are pushed down the traversal, and a list of open predecessors is returned up and passed on to the right.

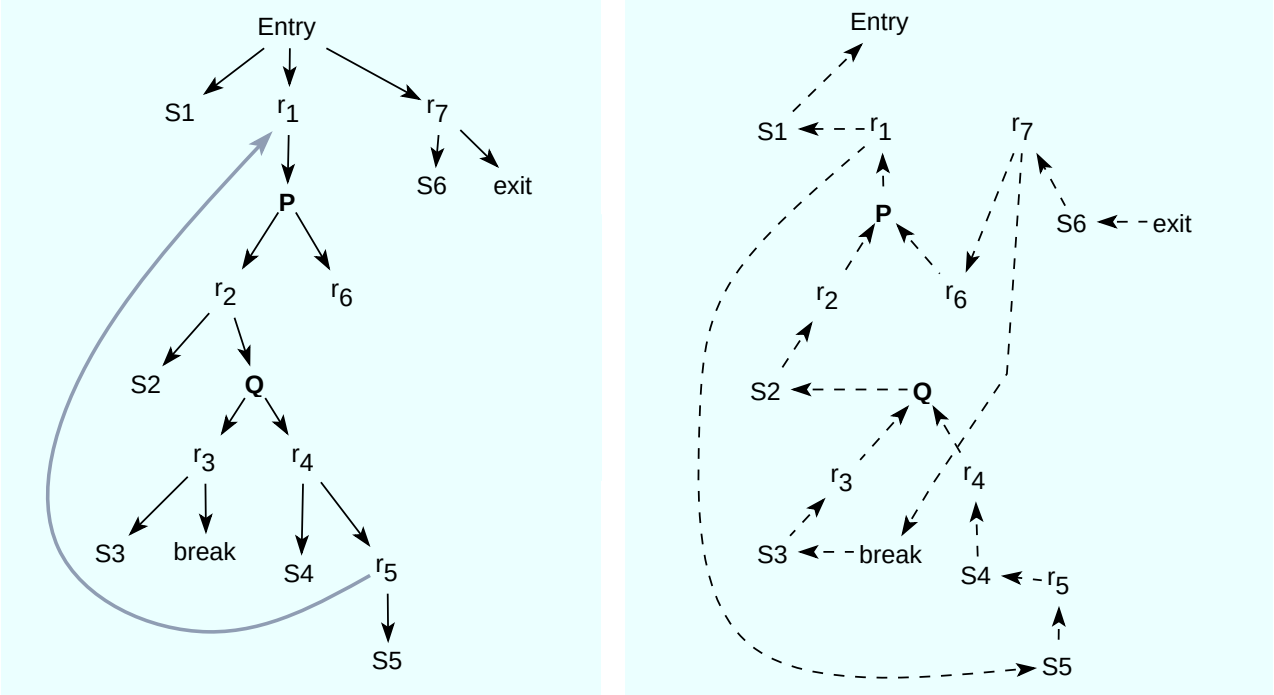


FIGURE 8. NQCDG and resulting reverse control flow graph for program in Figure 7

4.0 What about data flow analysis? Don't you still need a control flow graph?

Data flow analysis algorithms can be classified into two categories: iterative methods and elimination algorithms. Iterative methods propagate information through the control flow graph until the derived information converges. Elimination methods have two phases: the first phase operates by identifying and eliminating specific subcomponents of the control flow graph, while the second phase reconstitutes the graph and completes the propagation of derived information. Harrold and Malloy [7] present an iterative data flow analysis algorithm that operates on ordered control dependence graphs. In the remainder of this section, we adapt an elimination algorithm, Tarjan's interval analysis [8], [9], [10], [11], to use the control dependence graph.

Data flow analysis on CDGs requires that the CDG be ordered. If the CDG is constructed using the direct techniques presented in this paper or elsewhere, the proper ordering can be assured. If one is required to fall back to the general techniques, the algorithms provide no such assurance. For the remainder of this section, we assume that the given CDG is ordered.

4.1 Interval-Based Data flow Analysis using the CDG

While an iterative algorithm is useful, and perhaps necessary for handling irreducible programs, one can also adapt faster data flow analysis algorithms to the CDG. Tarjan's interval analysis algorithm is, perhaps, most natural. In Tarjan's approach, an interval corresponds to a strongly connected component in the control flow graph. Since a loop in the CFG is represented by a loop in the CDG, the adaptation of Tarjan's interval analysis to the CDG is fairly straightforward. The key definition in Tarjan's approach is that of the set *REACHUNDER*.

Definition 12 Let $CFG = \langle V, E \rangle$ be a control flow graph. Let x be a vertex in the CFG that is the target of a back edge. Then the interval whose header vertex is x is defined by the set $REACHUNDER(x) = \{x\} \cup \{y \in V$ such that y is not yet a member of any interval and x can be reached from y along a path not passing through x whose final edge is a back edge$\}$.

The following steps comprise Tarjan's algorithm for detecting intervals:

1. Create a depth-first spanning tree (DFST) of the CFG. Number the vertices sequentially according to their appearance in a preorder traversal. Mark back edges in the DFST at this time.
2. For each back edge $\langle v, x \rangle$ (in reverse of their preorder appearance) compute $REACHUNDER(x)$. Back edges are processed in reverse of their preorder occurrence to ensure that intervals will be detected from the inside out. Replace the vertices in $REACHUNDER(x)$ with a new virtual vertex denoting the interval whose elements are the vertices in *REACHUNDER(x)*.
3. If, in the computation of *REACHUNDER*, one reaches a vertex y that is not a descendant of x in the DFST, the graph is not reducible. Schwartz and Scharir [9] [10] provide a technique for isolating the irreducibility within an improper interval.
4. Once all back edges have been processed, the remaining vertices and intervals can be gathered into an outermost interval.

4.2 Relating Tarjan Intervals to the CDG

To show that the Tarjan intervals of the CFG can be easily discovered in the CDG, we first relate cycles in the CFG to cycles in the CDG.

Definition 13 Let G be a CFG in which every vertex is reachable. Given a path Π in G, a vertex n in Π is a *path exit predicate* iff no vertex in Π strictly postdominates n.

Because G contains no dead code, every cycle in the G has at least one path exit predicate.

Lemma 2 If Π is a path that begins with a path exit predicate w or Entry and ends with a path exit predicate v, then v is control dependent on w.

Proof: by induction on the length of Π.

Basis: let Π be a path of length 1. In this case, $v = w$ and w is trivially control dependent on itself.

Now assume the statement is true for all paths with length less than k, $k \geq 1$. Let Π be a path of length k. There are two cases:

1. Every vertex after w in Π is postdominated by v. In this case, v is immediately control dependent on w and we are done.
2. There is a vertex after w that is not postdominated by v.

Let x be the last vertex in Π that is not postdominated by v. Let $\Pi_{(x,v)}$ be the suffix of Π that starts with the last occurrence of x in Π. Let ${}_{(w,x)}\Pi$ be the prefix of Π that ends with the last occurrence of x in Π.

Claim 1: v is control dependent on x because (i) $\Pi_{(x,v)}$ is a nonnull path from x to v in which every vertex after x is postdominated by v (by our selection of x) and (ii) x does not strictly postdominate v (otherwise, v would not be a path exit predicate).

Claim 2: x is control dependent on w. Because there is a path from x to v and because v is not strictly postdominated by any vertex in Π, x is not strictly postdominated by any vertex in ${}_{(w,x)}\Pi$. As such, ${}_{(w,v)}\Pi$ is a path with a length less than k that starts and ends with a path exit predicate. By the inductive hypothesis this implies that x is control dependent on w.

Claims 1 and 2 imply that v is control dependent on w.

Theorem 5 There is a cycle in the CFG iff there is a cycle in the CDG.

Proof: ($\Leftarrow$) A cycle in the CDG implies a cycle in CFG. This follows directly from the definitions of control dependence.

Proof: ($\Rightarrow$) A cycle in the CFG implies a cycle in CDG. Let Π be a path that corresponds to a cycle in the CFG. Let x be the first (and last) vertex in Π. Let w be a path exit predicate for the path Π. Let ${}_{(x,w)}\Pi$ be the prefix of Π that ends with the first occurrence of w. Let $\Pi_{(w,x)}$ denote the suffix of Π that starts with the first occurrence of w. Consider the path ${}_{(x,w)}\Pi \cdot \Pi_{(w,x)}$. This is a path that meets the requirements of Lemma 2. As such, the vertex w is control dependent on itself and there must be a cycle in the CDG.

We can now state the definition of Tarjan intervals for a CDG.

Definition 14 Let H be a vertex in the CDG that is the target of a back edge. Then the interval whose header vertex is H, denoted *INTERVAL(H)* is defined by:

A region $R \in INTERVAL(H)$ if

1. R control depends upon H, and
2. There does not exist a region R' such that

   a.) R' is immediately control dependent upon R and

   b.) R' is control dependent on some region S that is not control dependent upon H.

A non-region vertex $N \in INTERVAL(H)$ if and only if it is immediately control dependent upon some region $R \in INTERVAL(H)$.

Part 1 of Definition 14 assures that each region in the interval may be executed if the loop header is executed. Parts 2(a) and 2(b) of Definition 14, taken together, assure that if control leaves the loop by other than the routine loop exit, the region containing the exit will not be added to the interval. Otherwise, control remains within the loop, and so that region is included in the interval.

As usual, once an interval in the graph has been identified, it can be replaced with a new vertex; the control dependence edges emanating from the new vertex will be copies of those that have sources inside the interval and targets outside the interval. Any control dependence edges which enter the interval must have the header region as their target.

In summary, the application of Tarjan's interval analysis to the control dependence graph is as follows:

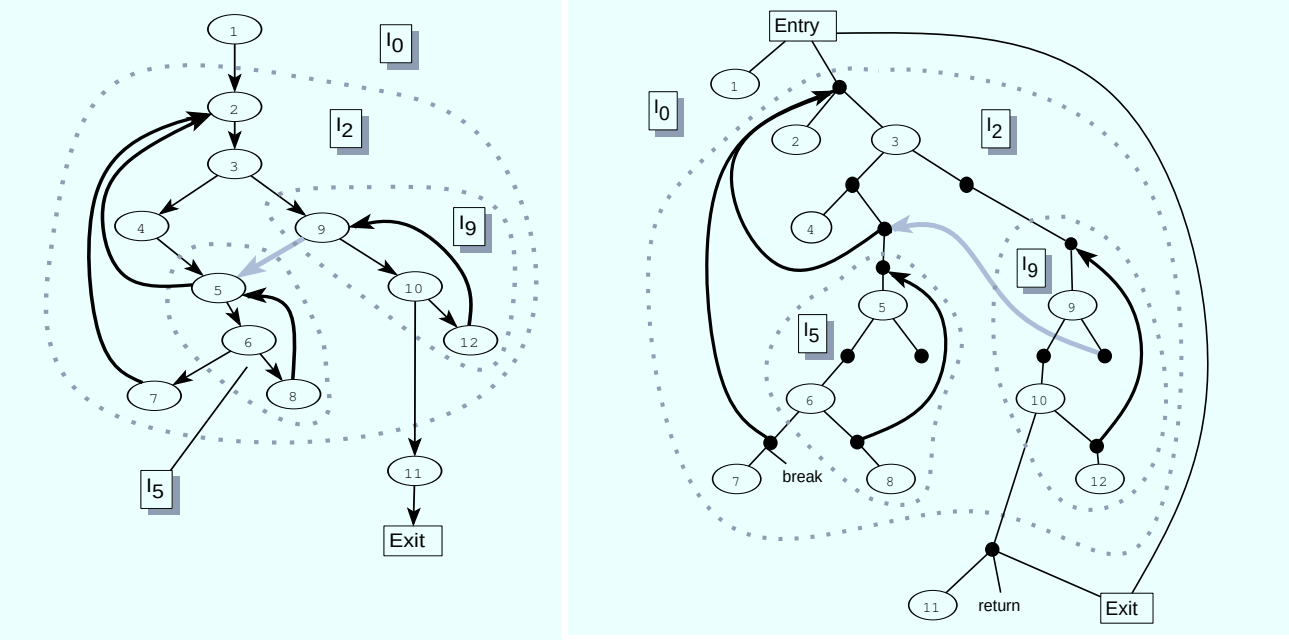
1. Create a depth-first spanning tree (DFST) of the CDG. Number the vertices sequentially according to their appearance in a preorder traversal. Mark back edges in the DFST at this time.
2. For each back edge $\langle v, x \rangle$ (in reverse of their preorder appearance) compute *INTERVAL*(x). Back edges are processed in reverse of their preorder occurrence to ensure that intervals will be detected from the inside out. Replace the vertices in *INTERVAL*(x) with a new virtual vertices denoting the interval whose elements are the vertices in *INTERVAL(x)*.
3. If, in the computation of *INTERVAL*, one reaches a vertex y that is not a descendant of x in the DFST, the graph is not reducible. The techniques of Schwartz and Scharir can be used to isolate the irreducibility within an improper interval.
4. Once all back edges have been processed, the remaining vertices and intervals can be gathered into an outermost interval.

Figure 9 illustrates the application of this algorithm. The control flow graph is that of Figure 24 of Ryder & Paull's survey [8]. The heavy gray edge in the CDG is a cross edge in the depth-first spanning tree of the CDG. Back edges in the CDG are shown as heavy solid arrows. Dotted lines indicate interval contours.

4.3 Reduction Order

Tarjan constructed a reduction order by imposing a two level numbering scheme (interval number, order in interval) where the interval number is the preorder number of the head of the interval, and the order of a vertex within an interval is defined by its position in a reverse postorder traversal of intervals (following edges in the DFST). In reduction order, intervals are processed in *reverse* of their preorder appearance to ensure that nested intervals are processed first. Similarly, the vertices within the interval are processed in reverse of postorder to ensure that all of the ancestors to a vertex are processed before the vertex itself.

When using the CDG, reduction order is even simpler. Reverse of preorder can still be used to ensure that nested intervals are processed before their enclosing intervals. Within an interval, however, a preorder traversal suffices to ensure that all ancestors to a vertex x are processed before x itself.



Control Flow Graph*

Control Dependence Graph*

*Outermost interval I_0 is not shown, but includes all vertices in the graph.

FIGURE 9. Example of Interval Analysis on CFG and CDG

4.4 Irreducible Graphs and Backward Data Flow Problems

Tarjan interval analysis on CDGs can easily be adapted to detect irreducibilities. In fact, the test mimics the test in the CFG: in computing *INTERVAL(x)*, if any of the CDG vertices have a preorder number that is not within the range $x \le y \le x + ND(x)$ where *ND(x)* is the number of descendants of *x*, or if any region *S* has a control dependence predecessor whose preorder number is not in that range, then the interval is improper. A number not in this range indicates that the vertex is not strictly a descendent (or strictly controlled by a descendent) of the head of the interval.

When an improper interval is detected, several strategies can be employed. One can revert to iterative techniques [7] entirely, one can node split to eliminate the improper interval, or one can apply the technique of Schwartz and Sharir [9], [10] to isolate the improper interval and apply iterative techniques only within the improper interval.

Tarjan [12] presents methods for efficiently computing both forward and backward data flow problems using path-compressed trees as the auxiliary data structure.

5.0 How often can I use these techniques?

A study of C source code indicates that, of over 119,000 functions, only 3572 (2.9%) contained `goto`s. The study involved examination of the BSD Unix kernel, the BSD

Unix commands, the X11 distribution, and portions of the Free Software Foundation's GNU distribution. Table 1 shows the basic data on the sources examined.

		BSD Kernel	BSD Commands	X	X Contributed	GNU[a]	Total
Number of Files	with gotos	359	974	134	448	150	2065
	without	1258	3940	1546	4910	533	12187
	Total	1617	4914	1680	5358	683	14252
Number of Functions	with gotos	771	1602	223	709	267	3572
	without	9022	26682	13896	60249	9896	119745
	Total	9793	28284	14119	60958	10163	123317
Lines		1671360		671628	2255312	703232	5301532

a. Including gdb, gdbm, bison, diff, emacs, fgrep, flex, gcc, gnuplot, grep, ispell, make, and RCS.

TABLE 1. Summary of sources examined in goto study

Of those `goto`s, 68% can be characterized as simple: a single label within the function reached by one or more `goto`s that branch to a target in the same compound statements or that branch outward from within control constructs. Simple forward branches can be interpreted as multilevel exit or break statements. Simple backward branches are looping constructs that often correspond to multilevel loop continuation. Simple `goto`s can be handled using the algorithms provided in this report. More complex patterns of usage will require fully general control flow analysis algorithms.

Complex patterns of branches include properly nested label-`goto` pairs (in which an outer pair branches over an inner pair), overlapping label-`goto` pairs (in which a `goto` branches to a label within a region spanned by another label-`goto` pair, and branches into control constructs. We were a bit dismayed to find that fully 10.5% of all `goto`s in the study branched into a control construct! Table 2 presents the summarized data.

			Number	Percentage
Simple	Single label in function	Forward	1662	24.6%
		Backward	866	12.8%
		Total	2528	37.4%
	Multiple labels in function	Forward	1650	24.4%
		Backward	405	6.0%
		Total	2055	30.4%
	Total		4583	67.8%
Complex	Nested		4	0.1%
	Overlapping		1461	21.6%
	Into control constructs		708	10.5%
	Total		2173	32.2%

TABLE 2. Summary of Goto usage found in sources

6.0 Acknowledgments

We wish to thank Barbara Ryder for several long and fruitful conversations concerning data flow analysis, and Brent McClure for his analysis of `goto` usage in C program. Mary Jeanne Harrold and Brian Malloy developed the iterative data flow analysis algorithms for CDGs; it was their use of ordered CDGs that first indicated the possibility of using the CDG directly for data flow analysis. This report has benefitted immensely from close readings by Suzanne Sluizer and Ksheerabdhi Krishna.

7.0 References

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

[2] R. A. Ballance, A. B. Maccabe, and K. J. Ottenstein, "The program dependence web: a representation supporting control, data, and demand-driven interpretation of imperative languages," *Proc. of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, New York, June 20–22, 1990, (appeared as Sigplan Notices, 25(6), June 1990), 257–271.

[3] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *TOPLAS*, 13(4), December 1991, 451–490.

[4] R. Cytron, J. Ferrante, and V. Sarkar, "Compact representations for control dependence," *Proc. of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, New York, June 20–22, 1990, (appeared as Sigplan Notices, 25(6), June 1990), 337–351.

[5] J. Ferrante, K. J. Ottenstein, and J. D. Warren "The program dependence graph and its use in optimization," *TOPLAS*, 9(3), October 1987, 319–349.

[6] M. J. Harrold and B. A. Malloy, "Data flow testing of parallelized code," *Proceedings of the Conference on Software Maintenance '92*, November, 1992 (to appear).

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

[8] B. G. Ryder and M. C. Paull, "Elimination algorithms for data flow analysis," *ACM Computing Surveys*, 18(3), September 1986, 277–316.

[9] J. T. Schwartz and M. Sharir, "A design for optimizations of the bitvectoring class," Courant Computer Science Report No. 17, Courant Institute of Mathematical Sciences, New York University, New York, September 1979.

[10] M. Sharir, "Structural analysis: A new approach to flow analysis in optimizing compilers," *Computer Languages*, 5, 1980, 141–153.

[11] R. E. Tarjan, "Testing flow graph reducibility," *J. Computer and System Sciences*, 9, 1974, pp: 355–365.

[12] R.E. Tarjan, "Fast algorithms for solving path problems," *JACM*, 28(3), July 1981, pp: 594–614.

8.0 Appendices

8.1 Sketch of CDG Construction Algorithm

This C++-like pseudo code illustrates the CDG construction algorithm in the absence of ill-behaved gotos. It assumes that each statement in the AST is represented as an instance of an object in the class that represents that particular statement type. Each of the major visit functions returns a boolean value which indicates whether the statement being visited had a normal ending. In this case, we interpret normal to mean that the statement (or nested statements) did not end with an explicit transfer of control.

```
cstack = new ConditionStack;
rstack = new RegionStack;

// This is the default behaviour for makeCDG
Bool
Stmt::makeCDG() {
  add self to rstack->top() region
  return TRUE;
}

// Specific statment behavior
Bool
IfStmt::makeCDG() {
  Region origTop = rstack->top();
  // Region(0) creates a new region with no predecessors
  Region pTrue = new Region(0);
  Region pFalse = new Region(0);
  Region pFollow = new Region(0);
  // Create predicate with origTop as control dependence predecessor, and
  // pTrue, pFalse as nominal successors.
  Region myRegion = new PredicateNode(origTop, pTrue, pFalse);

  // Visit the true part
  rstack->push(pTrue);
  cstack->push(IF_CON, pFalse, pFollow, 0);

  Bool normalEnding = thenPart->makeCDG();
  cstack->pop();
  // Pop region stack all the way back to where it was when this
  // procedure was invoked.
  rstack->popTo(origTop);

  if (elsePart->isPresent()) {
    rstack->push(pFalse);
    cstack->push(IF_CON, pTrue, pFollow, 0);
    normalEnding = elsePart->makeCDG() || normalEnding;
    cstack->pop();
    rstack->popTo(origTop);
  } else {
    if (! pFalse->hasSuccessors()) {
      remove pFalse from myRegion.
      delete pFalse;
    }
  }
  if (! normalEnding ) {
    // Need to check for non-normal exit in both legs
```

```
    // If pFollow has both pTrue and pFalse as predecessors, remove
    // both pFalse and pTrue as links and proceed as usual. If pFollow
    // has any other predecessors, it will become the new active region
    // on the stack.
  }
  if (pFollow->hasPredecessors() ) {
    rstack->push(pFollow);
  } else {
    delete pFollow;
  }
  return normalEnding;
}

Bool
WhileStmt::makeCDG() {
  Region origTop = rstack->top();
  // Constructor automatically makes pHdr control dependent uon origTop.
  Region pHdr = new Region(origTop);
  Region pTrue = new Region(0);
  // need to crate a pFalse region since it may end up being linked into
  // the CDG due to a loop exit downstream.
  Region pFalse = new Region(0);
  Region pFollow = new Region(0);
  Region myRegion = new PredicateNode(pHdr, pTrue, pFalse);

  // Visit the body
  rstack->push(pTrue);
  cstack->push(LOOP_CON, pFalse, pFollow, pHdr);

  Bool normalEnding = loopBody->makeCDG();
  if (normalEnding) {
    pHdr->addBackEdge(rstack->top());
  }
  cstack->pop();
  rstack->popTo(origTop);
  if (! pFalse->hasSuccessors() ) {
    delete pFalse;
  }
  if (! pFollow->hasPredecessors() ) {
    delete pFollow;
  }

  return normalEnding;
}

// DoStmt echoes whileStmt.
Bool
DoStmt::makeCDG() {
  Region origTop = rstack->top();
  Region pHdr = new Region(origTop);
  Region pContinue = new Region(pHdr);
  Region pTrue = new Region(0);
  Region pFalse = new Region(0);
  Region pFollow = new Region(0);
  Region myRegion = new PredicateNode(pContinue, pTrue, pFalse);

  // Visit the body
  rstack->push(pHdr);
  cstack->push(LOOP_CON, pFalse, pFollow, pContinue);
```

```
  Bool normalEnding = loopBody->makeCDG();
  if (normalEnding) {
    pHdr->addBackEdge(pTrue);
  }
  cstack->pop();
  rstack->popTo(origTop);
  if (! pFalse->hasSuccessors() ) {
    delete pFalse;
  }
  if (! pFollow->hasPredecessors() ) {
    delete pFollow;
  }
  return normalEnding;
}

Bool
BreakStmt::makeCDG()
{
  cstack->exitTo(LOOP_CON | SWITCH_CON);
  Region pFollow = newRegion(rstack->top());
  rstack->push(pFollow);
  // Pass back a non-normal ending
  return FALSE;
}

Bool
ReturnStmt::makeCDG()
{
  cstack->exitTo(ENTRY_CON);
  Region pFollow = newRegion(rstack->top());
  rstack->push(pFollow);
  // Pass back a non-normal ending
  return FALSE;
}

Bool
ContinueStmt::makeCDG()
{
  cstack->continueTo(LOOP_CON, rstack->top());
  Region pFollow = newRegion(rstack->top());
  rstack->push(pFollow);
  // Pass back a non-normal ending
  return FALSE;
}

Bool
StmtList::makeCDG()
{
  Bool normalEnding;
  Stmt s;

  for (s = first(); s != 0; s = next(s) ) {
    normalEnding = s->makeCDG();
  }
  return normalEnding;
}

// a goto has a <<label>> field
// To simplify life, we'll keep a table of <label, predecessor-list> pairs
// If we see a goto but have not seen the target, add the source of the
```

```
// goto to the table. Once we see the target, add any predecessors found
// in the table immediately.
Bool
gotostmt::makeCDG
{
  Region targetRegion = labelTable.findTarget(label);
  if (targetRegion) {
    targetRegion->addPredecessor(rstack->top());
  } else {
    labelTable.addTarget(label, rstack->top());
  }

  Region followRegion = new Region(rstack->top());
  rstack->push(followRegion);
  return FALSE;
}

// LabelledStmt: fields are <<label>> and <<stmt>>
Bool
LabelledStmt::makeCDG()
{
  Region currentRegion = rstack->top();
  Region myRegion = new Region(currentRegion);
  RegionList predList = labelTable.getPredecessors(label);
  if (predList) {
    myRegion->addPredecessorList(predList);
  }
  labelTable.addTarget(label, myRegion);
  rstack->push(myRegion);
  return stmt->makeCDG();
}
```

8.2 Handling the Condition Stack.

```
// Walking the construct stack
// The actual entry points are exitTo and continueTo.
// element(i) returns the ith element from the top of the stack.

// walkTo Returns the element index in the stack corresponding
// to the construct c
int
ConditionStack::walkTo(int eltIndex, Construct c)
{
  Element *e
  e = element(eltIndex);
  if (e.construct != c) {
    propagateLinks(eltIndex, c);
    e.followRegion -> addPredecessor(e.predRegion);
    return walkTo(eltIndex + 1, c);
  }
  return eltIndex;
}

ConditionStack::propagateLinks(int eltIndex, Construct c)
{
  int i = eltIndex;
  Element *e, *nextelt;
```

```
  e = element(i++);
  while (e.construct != c) {
    nextelt = element(i++);
    nextelt->followRegion -> addPredecessor(e->followRegion);
    e = nextelt;
  }
}

void
ConditionStack::exitTo(Construct c)
{
  walkTo(0, c);
}

void
ConditionStack::continueTo(Construct c, Region current)
{
  int index = walkTo(0, c);
  Element *e = element(index);
  e->hdrRegion -> addBackEdge(current);
}
```