Constructing Control Flow From Control Dependence

THOMAS BALL and SUSAN HORWITZ
University of Wisconsin – Madison

Control dependences characterize how the predicates in a program govern the execution of other statements and predicates. Control dependences are defined in terms of a program's control-flow graph; given a control-flow graph G, a corresponding control dependence graph, CDG(G), can be constructed. This paper addresses the inverse problem: we define an algorithm that, given a control dependence graph C, finds a corresponding control-flow graph G (*i.e.*, a graph G such that CDG(G) is isomorphic to C), or determines that no such control-flow graph exists. We call this process *CDG-reconstitution*.

CDG-reconstitution is a necessary part of systems that use and transform program dependence graphs (graphs that combine data dependences and control dependences) as the basis for discovering and performing code transformations. For example, parallelizing or vectorizing compilers as well as program integration tools may construct the program dependence graph(s) of a program (or set of programs) and then manipulate them in various ways to form a new graph. Reconstitution is needed as a final step to find a corresponding program for this graph. Such a process must respect the data and control dependences of the graph—we concentrate on the reconstitution problems caused by control dependences.

Our research provides two important advances over existing CDG-reconstitution algorithms. First, while previous algorithms handle only acyclic CDGs, our CDG-reconstitution algorithm correctly handles both acyclic and cyclic control dependence graphs. Second, our algorithm provides an improvement in efficiency over existing algorithms.

1. INTRODUCTION

Control dependences were introduced in [8, 10] to characterize how the predicates in a program govern the execution of other statements and predicates. Control dependences are defined in terms of a program's control-flow graph; given a control-flow graph G, a corresponding control dependence graph, CDG(G), can be constructed using the methods of [6, 7, 10]. This paper addresses the inverse problem: we define an algorithm that, given a control dependence graph C, finds a corresponding control-flow graph G (*i.e.*, a graph G such that CDG(G) is isomorphic to C), or determines that no such control-flow graph exists. We call this process *CDG-reconstitution*.

CDG-reconstitution is a sub-problem of *program* dependence graph reconstitution: given a program dependence graph (a graph that combines a program's control dependence and data dependence graphs), find a corresponding program or determine that no such program exists. Reconstitution of program dependence graphs is used in several areas:

(1) The program-integration algorithms of [14, 18, 22] combine portions of several program dependence graphs to form a merged graph. Program-dependence graph reconstitution is needed to determine the feasibility of the merged graph (the merged graph is feasible if it is the program dependence graph of some program) and, if it is feasible, to produce a corresponding program. Currently, these integration algorithms can handle only programs written in a language with very restricted control flow (for which CDG-reconstitution is trivial). The CDG-reconstitution algorithm presented in this

This work was supported in part by the National Science Foundation under grant CCR-8958530, by the Defense Advanced Research Projects Agency, monitored by the Office of Naval Research under contract N00014-88-K-0590, as well as by grants from Xerox, Kodak, and 3M.

Authors' address: Computer Sciences Department, Univ. of Wisconsin, 1210 W. Dayton St., Madison, WI 53706.

paper will permit the integration algorithms to be extended to handle a rich set of control constructs.

(2) Program-dependence graph reconstitution is also potentially useful in optimizing, vectorizing, or parallelizing compilers that perform source-to-source transformations by computing the program dependence graph of a program, performing transformations on this graph, and finally, finding a source program that corresponds to the transformed graph [5, 10].

Previously, [9, 11] have addressed the CDG-reconstitution problem, culminating in the results of [20], which addresses the issue of CDG-reconstitution in a formal framework. However, in this work, CDG-reconstitution is formalized only for acyclic CDGs (*i.e.*, the control dependence graphs of loop-free programs). An extension to handle cycles is mentioned briefly, but is incomplete and contains some errors. Our research provides two important advances. First, we define a single CDG-reconstitution algorithm that correctly handles both acyclic and cyclic control dependence graphs. Second, our algorithm provides an improvement in efficiency: in [20], the authors present an $O(N \times E)$ (where N and E are the number of vertices and edges, respectively, in the control dependence graph) algorithm for CDG-reconstitution when the control dependence graph is known beforehand to be feasible (*i.e.*, there is some corresponding control-flow graph) and an $O(N^2 \times E)$ algorithm for determining if a control dependence graph is feasible. Our algorithm takes an arbitrary control dependence graph C and either produces a corresponding control-flow graph or determines that C is infeasible in $O(N \times E)$ time; we also show how reconstitution can be done in linear time for large classes of CDGs.

Our algorithm is restricted in that it only considers the control dependence graphs of control-flow graphs that are reducible and for which no vertex inside a loop postdominates the loop's entry. This restriction greatly simplifies the process of reconstitution and allows for an efficient algorithm, while still providing an important improvement over the algorithm of [20], which handles an even more restricted class of control dependence graphs.

The main problem of CDG-reconstitution is to totally order the children of each predicate vertex in a way that is consistent with the postdomination ordering in a corresponding control-flow graph, if one exists. A feasible control dependence graph may have more than one corresponding control-flow graph. In this case, there is more than one total order that can be computed. Our CDG-reconstitution algorithm finds *one* corresponding control-flow graph for a given feasible control dependence graph by making some nondeterministic choices as explained below. Given a feasible CDG C, the algorithm determines, for all pairs of vertices (a,b) that have a common parent in C, which of the following holds:

(*i*) Vertex a postdominates vertex b in all corresponding control-flow graphs.

(*ii*) Vertex b postdominates vertex a in all corresponding control-flow graphs.

(*iii*) There is a corresponding control-flow graph in which vertex a postdominates vertex b, and a corresponding control-flow graph in which vertex b postdominates vertex a.

Pairs that fall into categories *(i)* and *(ii)* must be ordered $b \rightarrow a$ and $a \rightarrow b$, respectively; pairs that fall into category *(iii)* may be ordered nondeterministically as long as the resulting postdomination order over all the pairs is acyclic.

The remainder of this paper is organized as follows: Section 2 gives the basic definitions used throughout the paper; Section 3 presents our CDG-reconstitution algorithm and the ordering properties that are the basis of this algorithm; Section 4 considers related work; Section 5 examines some optimizations and extensions to CDG-reconstitution; Section 6 suggests directions for future research; the Appendix contains proofs of the ordering properties and proofs of correctness for the algorithms presented in Section 3. A glossary of terminology is included at the end of the paper, preceding the Appendix.

2. BACKGROUND

2.1. Control-flow Graphs and Control Dependence

A *control-flow graph* (CFG) is a directed, rooted graph that represents the flow of control among the portions of a program. A CFG has three types of vertices: statement vertices, which have one successor, predicate vertices, which have a *true*-successor and a *false*-successor, and an *EXIT* vertex, which has no successors. The root of the CFG is the *ENTRY* vertex, which is a predicate with *EXIT* as its *false*-successor and the first statement in the program as its *true*-successor. Finally, every vertex is reachable from the *ENTRY* and *EXIT* is reachable from every vertex.

Vertex w *postdominates* vertex v in a CFG iff $w \neq v$ and w is on every path from v to the *EXIT* vertex (sometimes written w **pd** v). Vertex w *immediately-postdominates* vertex v iff w **pd** v and there is no vertex x such that w **pd** x **pd** v (sometimes written w **impd** v). Vertex w *postdominates* the L-branch of predicate vertex v (where L is either *true* or *false*) iff w is the L-successor of v or w postdominates the L-successor of v. While no vertex can postdominate itself, a vertex can postdominate its own L-branch. The postdomination relation for a CFG is a tree with root *EXIT* where edge $v \rightarrow w$ denotes that v **impd** w.

Let G be a CFG and let v and w be vertices in G. Vertex v is an *L control dependence predecessor* of w (*i.e.*, w is directly control dependent on v—written $v \longrightarrow_c^L w$) iff w postdominates the L-branch of v and w does not postdominate v. A vertex can be directly control dependent on itself. Intuitively, if $v \longrightarrow_c^L w$, then whenever v executes and evaluates to L, w will eventually execute, barring abnormal termination such as an infinite loop or an exception. Furthermore, if v executes and does not evaluate to L, w might not execute. In this way, v directly controls whether or not w executes.

The control dependence graph (CDG) of a CFG is a directed graph that contains the same vertices as its CFG, excluding the *EXIT* vertex. The edges of the CDG are defined by the direct control dependence relation, as given above. The root of the CDG is the *ENTRY* vertex, from which every vertex in the CDG is reachable. Vertex v *dominates* vertex w in CDG C (written v **dom** w) iff every path from *ENTRY* to w contains v. Every vertex dominates itself. At times, we will also refer to domination in the CFG. However, unless explicitly stated otherwise, domination refers to the CDG.

Example. Figure 1 presents two CFGs, *G1* and *G2*, and their CDGs, *C1* and *C2*. In CFG *G1*, both vertices a and b postdominate the *true*-branch of r, but neither vertex postdominates r. Therefore, both a and b are *true* control dependent on r in *C1*. Vertex c is not control dependent on r because it postdominates r. In CFG *G2* both r and b postdominate the *false*-branch of q, but neither vertex postdominates q. Therefore, both r and b are *false* control dependent on q in *C2*.

A CFG G is a *corresponding CFG* of CDG C iff CDG(G) is isomorphic to C. The isomorphism must respect vertex text and edge labels.

2.2. The CDGs Under Consideration

The class of CDGs that is the focus of this work contains the CDGs of a *restricted class* of CFGs. A CFG is in this restricted class iff it is reducible[1] and for each vertex w that is a loop entry (*i.e.*, is the target of at

---

[1]A CFG G is *reducible* if the target of each back edge e of G (as defined by a depth-first search of G starting at the *ENTRY* vertex) dominates the source of e. Equivalently, G is reducible if any depth-first search of G identifies the same set of backedges. Roughly stated, reducibility restricts loops to be single-entry. Other equivalent characterizations of reducibility are given in [2].

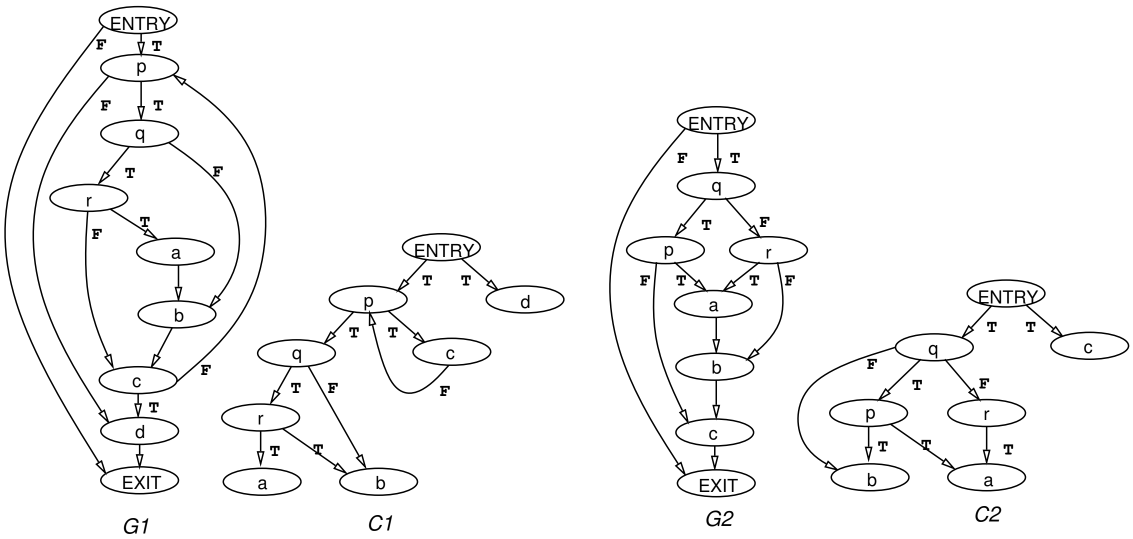


Figure 1. Control flow graphs *G1* and *G2* and their control dependence graphs *C1* and *C2*.

least one backedge), no vertex in the natural loop[2] of *w* postdominates *w*. The CDGs of such CFGs have certain properties that make reconstitution easier. Both CFGs *G1* and *G2* of Figure 1 fall into this class.

The class of CFGs described above properly contains the CFGs of programs written with **while** loops, sequencing, conditionals, and forward branches that do not jump into loops (this includes break and multi-level exit statements). In general, CFGs generated by **repeat-until** loops do not have the second property; vertices inside such loops can postdominate the loop entry.

As defined in the Introduction, a CDG *C* is *feasible* if it has a corresponding CFG. However, as our algorithm is only well-defined for the restricted class of CDGs described above, it may fail to reconstitute some feasible CDGs. To avoid confusion, we call a CDG *feasible* if it has a corresponding CFG; we call a CDG *R-feasible* it has a corresponding CFG *that is in the restricted class*. Similarly, we call a CDG *infeasible* if it is not feasible, and *R-infeasible* if it is not *R*-feasible. CDGs *C1* and *C2* of Figure 1 are *R*-feasible.

Finally, note that the goal of CDG-reconstitution is not to produce program text, but rather a CFG. A variety of syntactic constructs can give rise to the same CFG; the problem of determining or preserving syntactic constructs in reconstitution is not addressed here. A CFG or CDG may be annotated with information about the syntactic structures from which it was originally derived in order to recover the syntax later, or a structuring algorithm such as that of [3] may be employed to produce a program from a CFG.

2.3. The Difficulty of CDG Reconstitution

The CDG factors out the postdomination information that is present in the CFG; while the children of a given predicate vertex in a CDG are unordered, in the CFG these vertices are totally ordered by the

[2]Natural loops are defined in [2]. See the Appendix for a more detailed description of this restricted class of CFGs.

postdomination relation. The main problem of CDG-reconstitution is to recover an *acceptable* postdomination order solely from the examination of the CDG (*i.e.*, one that will produce a corresponding CFG). It might seem that the CDG could be annotated with the postdomination order from the CFG, thus eliminating this problem. However, the systems that use the CDG may transform it so that the original postdomination order is of no use to the reconstitution process (in fact, the original postdomination order may contradict the order imposed by a transformation).

To show why CDG-reconstitution is difficult, we first examine a class of CDGs for which it is simple. Consider a language that permits only sequencing, conditionals, and **while** loops (the CFGs generated by this language are a proper subset of the restricted class of CFGs). The CDG of a program in this language is essentially a tree in which **while** predicates are the *true* parents of the statements immediately enclosed in the loop, and **if** predicates are the *true* (*false*) parents of the statements immediately enclosed in the **then** (**else**) branch. The only non-tree edges are self-loops on **while** predicates. The CDG of a program in this language is nearly identical to the program's AST.

These observations give us a very simple and fast reconstitution algorithm for a CDG C that is composed only of tree edges and self-loop backedges: (1) pick an arbitrary total order for the L control dependence *tree* children of each predicate in the CDG; (2) let the "ordered" CDG represent an AST where predicates with self-loops are **while** loops, and predicates without self-loops are **if-then** or **if-then-else** conditionals, depending on whether or not they have *false* control dependence children. It is easy to see that no matter the order picked in step (1), the CDG of the program defined by the resulting AST is isomorphic to C.

Reconstitution is made difficult by the presence of forward, cross, and (true) backedges in the CDG. When such edges occur, it is not always possible to pick *any* total order for the L-children of a predicate in the CDG. In particular, the postdomination order of two vertices may be *fixed* for all corresponding CFGs. For example, in Figure 1, c **pd** q in every corresponding CFG of CDG $C1$, but the same does not hold true for vertices p and d. The difficulty of CDG-reconstitution is in distinguishing the cases where the postdomination order is fixed (and in what direction) from the cases where the postdomination order may be arbitrarily chosen.

3. A RECONSTITUTION ALGORITHM

The function ReconstituteCDG (see Figure 2) takes an arbitrary CDG C as input and behaves as follows:

(1) if C is not feasible then the function returns FAIL;

(2) if C is R-feasible then the function returns a corresponding CFG;

(3) if C is feasible but R-infeasible then the function may return FAIL or may return a corresponding CFG.

The algorithm performs three main steps. In line [1] the function ComputeGoodOrder is called. If C is R-feasible then this function returns, for each predicate vertex p in C, a total order for p's children that is consistent with the postdomination ordering in a corresponding CFG (for an R-feasible CDG, such an order is called a *good order*); the function may return FAIL if C is not R-feasible. In line [2] the function ConstructCFG is called to construct CFG G. The test in line [3] checks whether the CDG of G is isomorphic to C. This test is necessary because ComputeGoodOrder may not fail on some infeasible CDGs. If this test fails, then C is R-infeasible and ReconstituteCDG returns FAIL.[3]

[3]As presented here, the algorithm will succeed in reconstituting some CDGs that are feasible but not R-feasible. It is easy to modify the algorithm so that it succeeds iff the CDG is R-feasible: add a second test after line [3] that checks whether the CFG G is in the restricted class, returning FAIL if it is not.

function ReconstituteCDG(C : CDG) : CFG or FAIL
    Order : set of ordered pairs of vertices
    G : CFG
begin
[1] *Order* := ComputeGoodOrder(C)
    if *Order* = FAIL **then** return (FAIL) **fi**
[2] G := ConstructCFG(C, *Order*)
[3] **if** not Isomorphic(CDG(G),C) **then** return (FAIL) **fi**
    return (G)
end

Figure 2. Function ReconstituteCDG(C) produces a corresponding CFG for C if C is R-feasible.

The next three subsections present each of these three steps in greater detail. The final subsection gives a complete example of the reconstitution algorithm. Section 3.1 describes the main problem in reconstitution: how to compute a good order for a CDG. Section 3.2 describes how this order is used to construct a corresponding CFG. Finally, Section 3.3 illustrates the need for the CDG isomorphism check and describes how to perform it efficiently. Each of these sections includes a running time analysis. We show that the function ComputeGoodOrder takes $O(N \times E)$ time in the worst case, where N is the number of vertices in C and E is the number of edges in C, and that its running time dominates the running times for constructing the CFG and performing the isomorphism check. Therefore, the entire running time for function ReconstituteCDG is $O(N \times E)$ in the worst case.

3.1. Computing a Good Order for an R-feasible CDG

The first step of the CDG-reconstitution algorithm determines, for each predicate vertex p in the CDG, a total order for p's L-children that is consistent with the postdomination ordering in a corresponding CFG. For every pair of vertices (a,b) that have a common parent in the CDG (*i.e.*, $\exists\, p$ such that $p \longrightarrow_c^L a$ and $p \longrightarrow_c^L b$, where $L \in \{\, true,\ false \,\}$), the algorithm must decide whether a **pd** b or b **pd** a.[4] We refer to such pairs simply as (a,b) pairs.

By definition, a postdomination order for all (a,b) pairs in a CDG C is a *good order* for C if it is consistent with the postdomination ordering in a corresponding CFG. That is, *Order* is a good order for CDG C if there is a corresponding CFG G such that for all (a,b) pairs in C, *Order* specifies that a **pd** b iff a **pd** b in G (under the isomorphism map between C and CDG(G)). For example, a good order for CDG *C1* of Figure 1 is { b **pd** a, c **pd** q, d **pd** p }. An infeasible CDG has *no* good orders; a feasible CDG may have more than one good order.

The goal of this part of the CDG-reconstitution algorithm is to find a good order when the CDG is R-feasible. If the CDG is R-infeasible this step may fail or may return some acyclic order. Conceptually, a good order can be thought of as an acyclic graph where the vertices represent vertices from the CDG, there is either an edge $a \rightarrow b$ or $b \rightarrow a$ for every (a,b) pair, and edge $v \rightarrow w$ implies that w **pd** v in the good order.

[4]Consider vertices a and b such that $p \longrightarrow_c^L a$ and $p \longrightarrow_c^L b$. By the definition of control dependence, in every corresponding CFG of C both a and b postdominate the L-branch of p. This implies that in every corresponding CFG of C either a **pd** b or b **pd** a.

A good order (or any other acyclic ordering of all (a,b) pairs) totally orders the L-children of each predicate vertex in a CDG.

Unfortunately, we cannot always determine a good order simply by picking an arbitrary acyclic order over the (a,b) pairs of C.[5] The postdomination order of an (a,b) pair may be *fixed*; that is, it may be that a **pd** b in all corresponding CFGs or that b **pd** a in all corresponding CFGs. Once all fixed orders have been identified, a postdomination order for the remaining (a,b) pairs can be easily found.

Section 3.1.1 presents the ordering properties that form the basis for computing a good order. Section 3.1.2 refines one of these properties to allow for a more efficient algorithm. Section 3.1.3 makes use of these results to define a function ComputeGoodOrder that computes a good order for C if it is R-feasible.

3.1.1. Ordering Properties

This section introduces several ordering properties that can be used to determine a good order for an R-feasible CDG. The first two properties are about feasible CDGs: Property *OrderFixed* states that if a certain structural condition holds in a feasible CDG with respect to an (a,b) pair, then b **pd** a in all corresponding CFGs; Property *OrderArbitrary* states that if a different condition holds in a feasible CDG, then the postdomination order of the (a,b) pair is not fixed. Property *Complete* shows that for any (a,b) pair in an R-feasible CDG, either property *OrderFixed* or *OrderArbitrary* must hold (this is not true for feasible CDGs that are not R-feasible). Thus, for R-feasible CDGs, properties *OrderFixed* and *OrderArbitrary* can be stated in terms of "if-and-only-if", rather than "if-then." The proofs of these and other properties given in this section are included in the Appendix.

DEFINITION. A path in CDG C is *parent(a)-free* iff for all edges $v \longrightarrow_c^L w$ in the path, $v \longrightarrow_c^L a$ is not in C.

DEFINITION. Reachable(C,v) denotes the set of vertices reachable from v in CDG C (*i.e.*, $\{ x \mid v \longrightarrow_c^* x \}$).

PROPERTY *OrderFixed*
Consider an (a,b) pair in feasible CDG C. If there is a parent(a)-free path from *ENTRY* to a vertex in Reachable(C,b), then b **pd** a in every corresponding CFG of C.

The intuition behind property *OrderFixed* is as follows: if there is a parent(a)-free path from *ENTRY* to x, then there may be some execution of a corresponding CFG in which x executes at least once and a never executes. Suppose that $x \in$ Reachable(C,b) and that a **pd** b in some corresponding CFG G of C. By several properties of control dependence, a **pd** b and $x \in$ Reachable(C,b) imply that a **pd** x in G. Therefore, any time x is executed, a must eventually execute, which contradicts our claim that x may be executed once while a is never executed. Therefore, b **pd** a in all corresponding CFGs of C if there is a parent(a)-free path from *ENTRY* to a vertex in Reachable(C,b).

Example. Consider CDG *C1* in Figure 1: in this CDG, there is a parent(a)-free path from the *ENTRY* vertex to b: $ENTRY \longrightarrow_c^T p \longrightarrow_c^T q \longrightarrow_c^F b$. Therefore, b **pd** a in all CFGs corresponding to *C1*. Vertex p is in Reachable(*C1*, c) and $ENTRY \longrightarrow_c^T p$ is a parent(q)-free path. Therefore, c **pd** q in all CFGs corresponding to *C1*.

[5]As discussed in Section 2.1.3, this is possible when the CDG is a tree.

DEFINITION. DomReach(C) = { v | ∀ x ∈ Reachable(C,v): v **dom** x }.
Example. DomReach($C1$) = { *ENTRY*, p, d, q, a, b } and DomReach($C2$) = { *ENTRY*, q, c, a, b }

DEFINITION. Given an (a,b) pair in CDG C, b **over** a (by edge $v \longrightarrow_c^L b$) iff there exists $v \longrightarrow_c^L b$ such that (not $v \longrightarrow_c^L a$) and (not b **dom** v).
Example. In CDG $C1$ of Figure 1, b **over** a by edge $q \longrightarrow_c^F b$, and (not a **over** b).

PROPERTY *OrderArbitrary*
Consider an (a,b) pair in feasible CDG C. If not a **over** b, not b **over** a, and a,b ∈ DomReach(C), then there is a corresponding CFG in which b **pd** a, and a corresponding CFG in which a **pd** b.

Example. Consider CDG $C1$ in Figure 1. Since *OrderArbitrary* holds for the pair (p,d), there are at least two good orders for $C1$, one in which d **pd** p and one in which p **pd** d. In fact, there are exactly two good orders for this CDG, since property *OrderFixed* holds for every other (a,b) pair in $C1$.

In a feasible CDG, it is impossible for both property *OrderFixed* and *OrderArbitrary* to hold for an (a,b) pair. However, there are feasible CDGs for which neither property holds for some (a,b) pair. That is, these properties are not complete for feasible CDGs. However, they are complete for R-feasible CDGs:

PROPERTY *Complete*
If CDG C is R-feasible then for any (a,b) pair, either property *OrderFixed* holds (in one direction) or property *OrderArbitrary* holds.

The above three properties give us enough information to determine for which (a,b) pairs the postdomination order is fixed (and in what direction). The following property characterizes how to determine a good order for an R-feasible CDG:

PROPERTY *Permutations*
If CDG C is R-feasible, then an order for the (a,b) pairs of C is a good order iff it (1) respects the fixed pair orderings determined by property *OrderFixed* and (2) orders the (a,b) pairs for which *OrderFixed* does not hold according to an arbitrary total ordering of C's vertices.

3.1.2. Refining the Ordering Properties

A naive approach to check whether property *OrderFixed* holds for each (a,b) pair in a CDG will take worst-case O($N^2 \times E$) time, since there are at most O(N^2) pairs to check and we might have to traverse O(E) edges for each such pair to determine if there is a parent(a)-free or parent(b)-free path in the CDG. The following property permits a more efficient reconstitution algorithm by identifying a series of cheaper tests that are equivalent to property *OrderFixed* (when applied to R-feasible CDGs).

Every R-feasible CDG is reducible (see Appendix for proof). Thus, the set of backedges of an R-feasible CDG, as identified by any depth-first search, is uniquely defined (an edge $v \rightarrow w$ is a backedge of a reducible graph iff w **dom** v in the graph).

DEFINITION. Given reducible CDG C, the *backedge free* graph BF(C) is the graph obtained by removing all backedges from C.

PROPERTY *OrderFixed′*
Consider an (a,b) pair in R-feasible CDG C. b **pd** a in every corresponding CFG iff one of the following holds: **(a)** (not a **over** b) and (not b **over** a) and $b \notin$ DomReach(C). **(b)** (not a **over** b) and b **over** a. **(c)** (a **over** b) and (b **over** a by edge $v \longrightarrow_c^L b$ such that $v \longrightarrow_c^L x \longrightarrow_c^+ a$ in BF(C)).

Example. In CDG *C1* of Figure 1, case (b) of the property implies that b **pd** a and case (a) implies that c **pd** q. In CDG *C2*, case (c) implies that b **pd** a and case (b) implies that b **pd** r. The Appendix proves that properties *OrderFixed* and *OrderFixed′* are equivalent for R-feasible CDGs (for R-infeasible CDGs the properties may not be equivalent).

3.1.3. Function ComputeGoodOrder

We first describe an implementation of function ComputeGoodOrder that runs in worst-case $O(N \times E)$ time. Unfortunately, this algorithm may run in $O(N^2)$ expected time, even for CDGs that are trees. We then show how *region-identification* can improve the expected running time of this algorithm. Function ComputeGoodOrder is based directly upon the properties of the previous sections and performs the following steps:

(1) Preprocess the CDG C: check if C is reducible, and fail if not. Perform a depth-first search of C, classifying edges as tree, forward, cross, or back edges; compute DomReach(C). Section 3.1.3.1 shows how the set DomReach(C) can be computed in $O(N+E)$ time.

(2) For each (a,b) pair in C, determine if property *OrderFixed′* holds (and in which direction).

(3) Arbitrarily number the vertices of the CDG to order each (a,b) pair for which property *OrderFixed′* does not hold.

(4) If there is a cycle in the pairwise orderings of the (a,b) pairs then fail (if C is R-feasible, no such cycle can arise). If C is R-feasible, the resulting order will be a good order.

We now consider the running time of this algorithm. Step (1) requires linear time since checking for reducibility is a linear-time operation [21], as is depth-first search and computing DomReach(C). Step (2) considers every (a,b) pair in C (possibly $O(N^2)$ of them) in turn. To compute whether or not property *OrderFixed′* holds for an (a,b) pair, the following three operations are needed:

- Membership in DomReach(C). Vertex membership in this set can be performed in constant time with a boolean test.

- Determining if b **over** a. This is a worst-case $O(N)$ operation since a vertex may have as many as N predecessors. In practice, the number of predecessors of any vertex is bounded by a small constant and this operation takes constant time. The CDG domination relation is not required to compute b **over** a. Given a candidate edge $v \longrightarrow_c^L b$ (for b **over** a), b **dom** v iff $v \longrightarrow_c^L b$ is a backedge (as identified by the depth-first search of the CDG).

- Determining if $v \longrightarrow_c^L x \longrightarrow_c^+ a$. Computing this transitive closure can take $O(E)$ time, which implies a worst-case overall running time of $O(N^2 \times E)$ for step (1). However, by caching the results of each closure operation, subsequent occurrences of this case may take constant time. In the worst case, amortizing over all (a,b) pairs, a transitive closure will have to be computed from each vertex in C, requiring $O(N \times E)$ time.

From the above analysis, it should be clear that the worst-case running time of step (2) is $O(N \times E)$, which dominates $O(N^2)$ and is equivalent (in the worst case) to $O(N^3)$. Step (3) requires $O(N)$ time and step (4), checking for a cycle, is a worst-case $O(N^2)$ operation since there may be $O(N^2)$ ordered pairs (*i.e.*, the ordering corresponds to a graph with at most N vertices and N^2 edges—a graph with M vertices and F edges can be checked for cycles using depth-first search in time $O(M+F)$). Thus, the worst-case running time for function ComputeGoodOrder is $O(N \times E)$.

Unfortunately, the expected running time of the above algorithm may be $O(N^2)$, even for CDGs that are trees. Consider the case of a tree of height two that has N leaves. As discussed in Section 2.3, CDGs that are trees can be processed in linear time. However, the above algorithm will examine $O(N^2)$ pairs. This time bound can be improved if we assume, as done in [20], that *regions* already have been identified in the input CDG. Regions partition the vertex set of a CDG, grouping vertices with common sets of control parents. Regions are identified in BF(C) rather than in C. Vertices a and b are in the same region iff

$$\{ (v,L) \mid v \longrightarrow_c^L a \text{ in BF}(C) \} = \{ (v,L) \mid v \longrightarrow_c^L b \text{ in BF}(C) \} \neq \emptyset.$$

For reducible CDGs, this is equivalent to: a and b are in the same region iff they are an (a,b) pair, (not a **over** b), and (not b **over** a).

Suppose that region R contains the vertices $\{ x_1, \cdots, x_n \}$. Note that only case (a) of property *OrderFixed′* can apply to any pair of vertices from R. Furthermore, if C is R-feasible then there is at most one x_k such that $x_k \notin$ DomReach(C) (otherwise a cycle would arise in the postdomination order). This implies that if no such x_k exists then property *OrderArbitrary* holds for all pairs in R, and that if such an x_k exists then it must postdominate all other vertices in R and property *OrderArbitrary* holds for all the other pairs in R. We need only examine each vertex in a region once to determine the order for *all* the (a,b) pairs in the region.

However, an (a,b) pair may span two regions (*i.e.*, a and b may be in different regions, R_1 and R_2). It is impossible for property *OrderArbitrary* to hold for such an (a,b) pair. In this case, either case (b) or (c) of property *OrderFixed′* will determine a postdomination order for the (a,b) pair. Furthermore, it is easy to show that for any two (a,b) pairs (call them (c,d) and (e,f)) such that c and e are in R_1 and d and f are in R_2, property *OrderFixed′* implies that d **pd** c iff property *OrderFixed′* implies that f **pd** e (and *OrderFixed′* implies that c **pd** d iff *OrderFixed′* implies that e **pd** f). Thus, to order all the (a,b) pairs that span the same two regions requires only one application of property *OrderFixed′*.

Thus, to process all (a,b) pairs that reside in the same region takes $O(N)$ time. If no (a,b) pairs span a region then function ComputeGoodOrder runs in linear time. By an analysis similar to that for step (2) of function ComputeGoodOrder, processing all (a,b) pairs that span different regions takes $O(R \times E)$ time, where R is the number of regions in the CDG. In the worst case, $R = N$. However, in practice, there are many fewer regions than vertices in CDGs.

3.1.3.1. Computing DomReach(C)

This section precisely characterizes when a vertex v dominates all the vertices in Reachable(C,v), and then shows how to compute this information efficiently for all vertices in C using a depth-first search.

A depth-first search of C partitions its edge set into four classes: tree edges (TE), forward edges (FE), back edges (BE), and cross edges (CE). The preorder number of vertex v is denoted by $\#v$. Vertex v is an *ancestor* of vertex w iff $v = w$ or there is a sequence of tree edges from v to w. Vertex v is a *proper ancestor* of vertex w iff $v \neq w$ and v is an ancestor of w. Given distinct vertices a and b, let lca(a,b) be the least-common ancestor of a and b in the spanning tree (TE) of graph C.

LEMMA. Vertex v dominates all vertices in Reachable(C,v) iff all the following hold:

(1) there is no forward edge $a \to b$ such that a is a proper ancestor of v and v is a proper ancestor of b;

(2) there is no back edge $b \to a$ such that a is a proper ancestor of v and v is an ancestor of b;

(3) there is no cross edge $a \to b$ such that v is a proper ancestor of b or a, and lca(a,b) is a proper ancestor of v.

PROOF. The proof is straightforward and is left to the reader. □

To determine when the condition in the above lemma holds, a LOW numbering, similar to that used to determine the biconnected components of undirected graphs [1, pg. 179], is defined for each v in C:

$$\begin{aligned}\mathrm{LOW}[v] = \mathrm{MIN}(\; & \{\#v\} \\ \cup\; & \{\#a \mid \exists\, a \to b \in FE \text{ such that } v \text{ is a proper ancestor of } b\,\} \\ \cup\; & \{\#a \mid \exists\, b \to a \in BE \text{ such that } v \text{ is an ancestor of } b\,\} \\ \cup\; & \{\#\mathrm{lca}(a,b) \mid \exists\, a \to b \in CE \text{ such that } v \text{ is a proper ancestor of } a \text{ or } b\,\}\;)\end{aligned}$$

By the above lemma, v dominates all vertices in Reachable(G,v) iff LOW[v] $\geq \#v$. The calculation of LOW[v] can be embedded in a depth-first search of v by rewriting the LOW equation for a vertex v in terms of the LOW values for v's children in the tree TE. The rewritten equations are:

$$\begin{aligned}\mathrm{LOW}[v] = \mathrm{MIN}(\; & \{\#v\} \\ \cup\; & \{\#a \mid \exists\, v \to a \in BE\,\} \\ \cup\; & \{\,\mathrm{UP}[s] \mid v \to s \in TE\,\}\;)\end{aligned} \qquad \begin{aligned}\mathrm{UP}[v] = \mathrm{MIN}(\; & \mathrm{LOW}[v] \\ \cup\; & \{\#a \mid \exists\, a \to v \in FE\,\} \\ \cup\; & \{\#\mathrm{lca}(a,b) \mid \exists\, a \to b \in CE \text{ such that } v = a \text{ or } v = b\,\}\;)\end{aligned}$$

Using the rewritten equations, LOW[v] can be computed for all v in G via a postorder traversal of the spanning tree TE, an $O(N)$ operation. Each operation done during the computation of LOW is a constant time operation, given that the least common ancestor of each cross edge has been computed beforehand. An algorithm for computing least common ancestors in trees by Harel and Tarjan [12] takes $O(N+M)$ time, where N is the number of vertices in the tree and M is the number of queries (in this case $M = |CE|$).

3.2. Constructing the Control-Flow Graph

This section shows how to construct a corresponding CFG from a feasible CDG C and a good order *Order*. For each predicate p with L-children in C, the L-children of p are totally ordered by the postdomination information supplied by *Order*. Let the list $CLIST(p, L) = (v_1, \ldots, v_m)$ be the L-children of p ordered so that for all i, $1 \leq i < m$, the ordered pair (v_i, v_{i+1}) is in *Order* (*i.e.*, *Order* specifies that v_{i+1} **pd** v_i).

The following rules describe how CFG postdomination information can be propagated in a top-down fashion over the CDG and used to determine the edge set of the desired CFG G (a CFG that corresponds to C and respects *Order*):

- The immediate postdominator of the *ENTRY* vertex in any CFG is the *EXIT* vertex.
- If vertex w immediately follows v in CLIST(p,L), then w **impd** v in G.
- If vertex v occurs last in CLIST(q,L) and w **impd** q in G, then w **impd** v in G.

These rules provide the basis for the function ConstructCFG of Figure 3, which first builds the CLISTs and then calls the procedure DFS to produce the edge set of the CFG via a depth-first search of C. The first parameter to DFS (v) is the current vertex of the depth-first search and the second parameter (w) is always the immediate postdominator of v, which is updated in accordance with the above rules. The edges of the control-flow graph are determined as follows:

- The CFG successor of a statement vertex is its immediate postdominator.
- The CFG L-successor of a predicate p such that CLIST(p,L) ≠ () is the first vertex in CLIST(p,L).
- The CFG L-successor of a predicate p such that CLIST(p,L) = () is p's immediate postdominator.

If C is feasible and *Order* is a good order for C, then function ConstructCFG will return a *corresponding* CFG for C. Otherwise, ConstructCFG will return a CFG that does not correspond to C.

```
global
  CLIST : (vertex, LABEL) → list of vertices;
  Edges : set of edges


function ConstructCFG(C:CDG,
   Order: set of ordered pairs of vertices) : CFG
begin
   CLIST := ConstructCLIST(C,Order)
   Edges := ∅
   unmark all vertices in C
   DFS(ENTRY,EXIT)
   return( (vertices(C) ∪ {EXIT}, Edges) )
end

procedure DFS( v,w : vertices );
 begin
   if not Marked(v) then
      Marked(v) := TRUE;
      if v is a statement then Edges := Edges ∪ { v → w }
      else  /* v is a predicate */
         for LBL := false to true do
            if CLIST(v,LBL) = () then
               Edges := Edges ∪ { v →^LBL w }
            else
               curr := head(CLIST(v,LBL));
               rest := tail(CLIST(v,LBL));
               Edges := Edges ∪ { v →^LBL curr }
               while rest ≠ () do
                  DFS(curr,head(rest))
                  curr := head(rest); rest := tail(rest)
               od
               /* curr is now the last vertex in the CLIST */
               DFS(curr,w)
            fi
         od
      fi
   fi
 end
```

Figure 3. Function ConstructCFG uses a depth-first search of the CDG C and the order information from *Order* to generate the edges for the CFG.

A topological sort of the vertices of C (using the pair ordering of *Order*) is all that is required to construct the CLISTs. This takes worst-case $O(N^2)$ time since *Order* contains, in the worst case, $O(N^2)$ pairs (this can be improved by use of region-identification, in much the same way function ComputeGoodOrder can be improved). The depth-first search to generate the edges of the CFG takes $O(E)$ time.

3.3. Final Isomorphism Test

Once step [3] of function ReconstituteCDG (Figure 2) is reached, a CFG G has been constructed. If C is R-feasible then G corresponds to C. Furthermore, CDG(G) is guaranteed to be isomorphic to C via a mapping constructed by uniquely tagging the vertices of C and carrying the tags over to the vertices of G and CDG(G) (see the Appendix for proof). We can check whether the mapping provided by these tags is an isomorphism map between C and CDG(G) in time $O(N+E)$.

If C is not R-feasible, then either the CDG of G is not isomorphic to C or G is not in the restricted class of CFGs. Figure 4 illustrates these two points. The CDG shown in Figure 4(a) is infeasible. However, function ComputeGoodOrder will not fail and will return an order in which b **pd** a (case (a) of property *OrderFixed′* implies that b **pd** a). Function ConstructCFG will build the CFG as shown. The CDG of this CFG is not isomorphic to the input CDG.

The CDG shown in Figure 4(b) is feasible but R-infeasible (neither property *OrderFixed* nor property *OrderArb* holds for the (a,b) pair). However, case (a) of property *OrderFixed′* implies that b **pd** a; therefore, assuming that case (a) of property *OrderFixed′* is tested first, function ComputeGoodOrder will return an order in which b **pd** a, which happens to be a good order for the CDG. Function ConstructCFG will build a corresponding CFG. This CFG is not in the restricted class because b postdominates loop entry a.

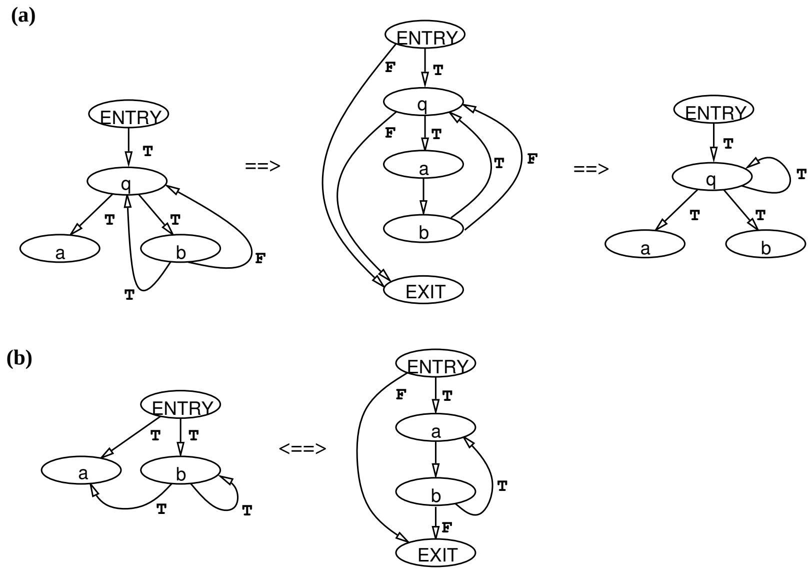
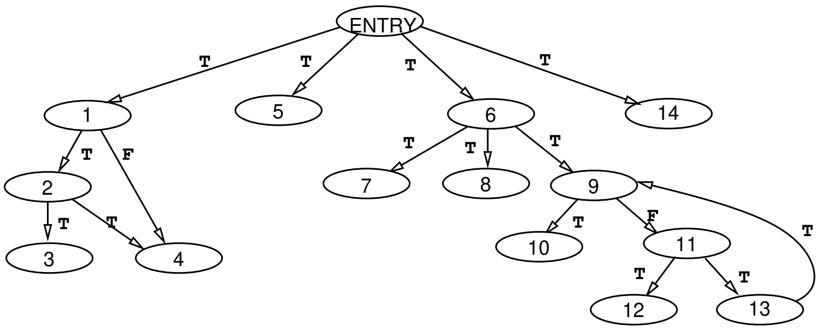


Figure 4. (a) An infeasible CDG and (b) an R-infeasible CDG.

3.4. Putting It All Together

This section illustrates how CDG-reconstitution produces a CFG from a CDG. Consider the *R*-feasible CDG in Figure 5(a). Property *OrderFixed* only holds for two (a,b) pairs; it determines that 4 **pd** 3 and 13 **pd** 12. In terms of property *OrderFixed′*, 4 **pd** 3 because 4 **over** 3 and (not 3 **over** 4), and 13 **pd** 12 because (not 12 **over** 13), (not 13 **over** 12), and $13 \notin$ DomReach(C). Property *OrderArbitrary* holds for every other (a,b) pair. Thus, we are free to give any total order to the *T*-children of *ENTRY* and to the *T*-children of vertex 6. Note that $(10,11)$ is not an (a,b) pair because 10 and 11 do not share a common *L*-parent.

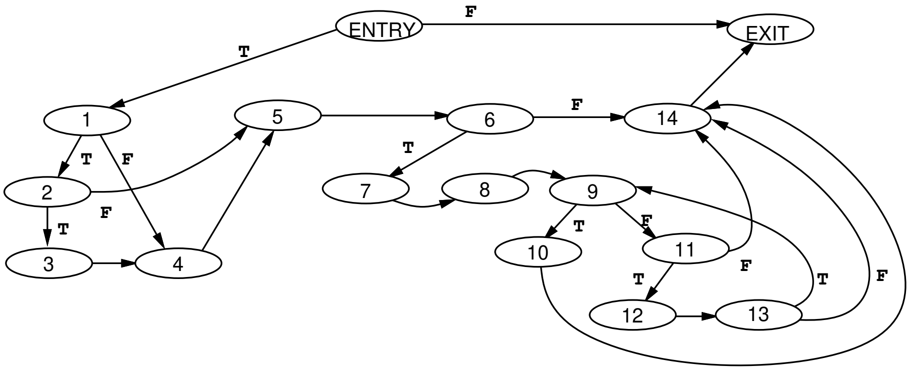
Figure 5(b) gives a good order for the CDG. To make the example easier to explain, we choose an order that matches the layout of the CDG in 5(a). We emphasize that there are other good orders for this CDG.



(a)

4 **pd** 3   13 **pd** 12   14 **pd** 6 **pd** 5 **pd** 1   9 **pd** 8 **pd** 7

(b)



(c)

Figure 5. (a) An *R*-feasible CDG; (b) a good order for the CDG, and (c) the corresponding CFG that would be constructed by function ConstructCFG.

The graph in 5(c) is the corresponding CFG that results from applying function ConstructCFG to the CDG and the good order. We illustrate how a portion of the CFG is built. Here are some of the CLISTs and a portion of the execution of function DFS (see Figure 3), which creates the edge set of the CFG:

Some CLISTs		
CLIST(*ENTRY*,*F*)	=	()
CLIST(*ENTRY*,*T*)	=	(1,5,6,14)
CLIST(1,*F*)	=	(4)
CLIST(1,*T*)	=	(2)
CLIST(2,*F*)	=	()
CLIST(2,*T*)	=	(3,4)

Execution of DFS	
[1]	DFS(*ENTRY*,*EXIT*)
[2]	add edge $ENTRY \rightarrow^{F} EXIT$
[3]	add edge $ENTRY \rightarrow^{T} 1$
[4]	DFS(1,5)
[5]	add edge $1 \rightarrow^{F} 4$
[6]	DFS(4,5)
[7]	add edge $4 \rightarrow 5$
[8]	add edge $1 \rightarrow^{T} 2$
[9]	DFS(2,5)
[10]	add edge $2 \rightarrow^{F} 5$
[11]	add edge $2 \rightarrow^{T} 3$
[12]	DFS(3,4)
[13]	add edge $3 \rightarrow 4$
[14]	DFS(4,5)
[15]	DFS(5,6)
	....

Recall that the second vertex in the call to DFS is always the immediate postdominator of the first. DFS is initially invoked (line [1]) with the call DFS(*ENTRY*,*EXIT*). Since CLIST(*ENTRY*,*F*) is empty, the edge $ENTRY \rightarrow^{F} EXIT$ is made (line [2]). The edge $ENTRY^{T} \rightarrow 1$ is added because 1 is the first vertex in CLIST(*ENTRY*,*T*) (line [3]). Since 5 follows 1 in CLIST(*ENTRY*,*T*), 5 is passed down as the immediate postdominator of 1 in the call DFS(1,5) (line [4]). The edge $1 \rightarrow^{F} 4$ is added because 4 is the first vertex in CLIST(1,*F*). Since 4 is also the last vertex in CLIST(1,*F*), 5 is passed down as the immediate postdominator of 4 in the call DFS(4,5) (line [6]). Because 4 is a statement vertex, the edge $4 \rightarrow 5$ is added (line [7]). At this point, we return to process CLIST(1,*T*) in the active call DFS(1,5).

4. RELATED WORK

As discussed in the Introduction, the most important previous work on CDG-reconstitution is that of [20]. In this section we discuss the similarities and differences between that work and the work presented here.

The major difference between our work and that of [20] is in the treatment of loops. In [20], CDG-reconstitution is formalized only for control dependence DAGs as opposed to general cyclic CDGs. An extension to handle cycles is mentioned briefly, but is incomplete and contains some errors, which are corrected in our CDG-reconstitution algorithm. For example, in the control dependence DAG representation of [20], CDG *C1* of Figure 1 would not include the backedge from *c* to *p*. The graph would be augmented by marking *c* as an exit vertex of the loop headed by *p* and marking *p* as a loop entry. However, the ordering rules of [20] do not force *c* to postdominate *q*, which it must; exit vertices must always postdominate other vertices with the same control dependence parents.

Considering CDG *C1* of Figure 1 again, according to [20], p should be treated as a special *forall* vertex. According to their ordering rules, if a forall vertex and a statement vertex have a common control parent, the forall vertex must follow the statement vertex in every corresponding CFG. Thus, p would be forced to follow d; however, the order of these two vertices is not fixed. Forcing p to follow d would be a problem if there were a flow dependence from a to d, requiring that d follow p.

Another difference between our approach and that of [20] is that they present two separate algorithms, one to test feasibility and the other to produce a corresponding CFG for a given feasible CDG, while we define a single algorithm that either produces a corresponding CFG, or determines that the given CDG is infeasible. Their feasibility-testing algorithm is $O(N^2 \times E)$, and their CDG-reconstitution algorithm is $O(N \times E)$. Our (single) algorithm is $O(N \times E)$.

Two aspects of the work of [20] that appear different but that are actually only superficially so, have to do with the class of CDGs that can be handled. Both the algorithm presented in this paper and the algorithms of [20] handle restricted classes of feasible CDGs. In [20] these restrictions are defined directly in terms of properties of the CDGs. In contrast, we define the restrictions *indirectly* by saying that we handle only those CDGs that correspond to a restricted class of CFGs. Our algorithm handles a superset of the CDGs considered in [20].

Another aspect of the question of what CDGs are handled has to do with *region* vertices (also called *forall* vertices), which are assumed in [20] to have been added to the CDG (region vertices are added to a CDG to gather all vertices with the same set of control conditions together). As discussed in Section 3.1.3, our algorithm is easily extended to handle CDGs with regions.

5. OPTIMIZING AND EXTENDING CDG-RECONSTITUTION

This section examines how to improve the efficiency of CDG-reconstitution for commonly occurring classes of CDGs, and considers the problems involved in extending CDG-reconstitution to handle a larger class of CDGs.

5.1. Improving Efficiency

There are several ways in which the efficiency of our CDG-reconstitution algorithm can be improved by using a prepass over the CDG to identify and exploit certain regular structure in the CDG.

If the given CDG corresponds to a program written with only while loops, (multi-level) exit, and if-then-else statements, then CDG-reconstitution takes time $O(N+E)$, as opposed to $O(N \times E)$. The CDGs corresponding to such programs are graphs whose edge sets can be partitioned into a set of tree edges and a set of backedges via a depth-first search (*i.e.*, there are no forward or cross edges). To order the L-children of a predicate p we merely scan the children to see if there is a child y such that $y \notin$ DomReach(C) or $p \longrightarrow_c y$ is a backedge. If C is R-feasible then vertex y must postdominate all other L-children of p. The other L-children of p can be ordered arbitrarily with respect to one another.

When a vertex is a member of DomReach(C), the CDG subgraph reachable from that vertex can be "clipped" out of the CDG, resulting in two CDGs. CDG-reconstitution can be done on each graph independently and the results combined. This has two potential benefits. First, it can reduce the cost of transitive closure needed in case (c) of property *OrderFixed′*. Second, if the clipped subgraph corresponds to a well-structured program, the subgraph can be processed in linear time, as noted above. As shown in Section 3.1.3.1, the set DomReach(C) can be computed in linear time.

5.2. Extending CDG-Reconstitution

Our CDG-reconstitution algorithm can only be applied successfully to the CDGs corresponding to a restricted class of CFGs. This section presents two ways to extend the reconstitution algorithm to handle the CDGs of all reducible CFGs (handling the CDGs of irreducible CFGs adds an additional level of complexity, but such CFGs hardly ever arise in practice). We first examine the difficulties in extending CDG-reconstitution directly to CDGs of reducible CFGs with loops that do not satisfy the postdomination property (we refer to these loops as *repeat* loops and to loops with the postdomination property as *while* loops), and then consider an approach that sidesteps the reconstitution problem by transforming the input CFG.

CDG-reconstitution is much harder for CDGs that correspond to CFGs containing repeat loops because these CDGs are not as well-structured as the CDGs of CFGs in the restricted class. Vertices that are inside a repeat loop of a CFG may be directly control dependent on vertices outside the loop; in contrast, vertices that are inside a while loop can only be directly control dependent on the loop entry or other vertices within the loop. This lack of structure complicates the definition of property *OrderFixed*—there are many more cases that determine when the postdomination order is fixed. We have a candidate set of such conditions, but are not sure that we have completely identified them. Although we have a property *OrderArbitrary* for this larger class of CDGs, more work is needed to find a suitable *OrderFixed* property and a way to compute it efficiently for the larger class of CDGs.

Although, theoretically, the CDG-reconstitution algorithm presented here works only for CDGs of CFGs in the restricted class, in practice, we believe that a simple CFG transformation will make this algorithm applicable to the CDGs of any reducible CFG. Given a reducible CFG, natural loop analysis [2] can be used to identify the nested loops in the graph. Those loops that cannot be expressed as while loops (loops containing a vertex that postdominates the loop entry) can be rewritten in terms of a while loop (that has the postdomination property) and specially marked. As long as the manipulations that are applied to the CDGs of such CFGs maintain the feasibility of the CDG and do not introduce structures that correspond to repeat loops, the CDG-reconstitution algorithm given here will succeed in reconstituting the CDG.

A CFG constructed by our CDG-reconstitution algorithm will only contain loops that are expressible as while loops. The inverse of the repeat-to-while loop transformation can be applied to any loop that was specially marked. This entire process is independent of the control constructs in the source language that generated the original CFG.

6. FUTURE WORK

6.1. Data Dependences and PDG-reconstitution

CDG-reconstitution is a component of PDG-reconstitution. In PDG-reconstitution, data dependences (as well as control dependences) determine an order between vertices. We have described and proved correct an algorithm for PDG-reconstitution for PDGs in which the underlying CDG is a tree [4]. A data dependence from vertex v to w will determine an order between the ancestors of v and w in the CDG that are children of the least-common ancestor of v and w. In a CDG that is a general graph, the same general principle holds (by ignoring the backedges in the CDG, the least-common ancestor relation is well-defined). We believe that the algorithm in [4], incorporated with the techniques described in this paper, will easily extend to PDGs with complex control dependence.

The complexity of PDG-reconstitution depends on the types of data dependences represented in the PDG. In the PDGs considered in [4], data dependences include loop-independent and loop-carried flow edges, and def-order edges (which are similar to output dependences), but not anti-dependences. Because there are fewer ordering constraints in these PDGs (than in a PDG that includes anti and output dependences), PDG reconstitution is algorithmically and computationally more complex.

6.2. A Semantic Justification of Control Dependence

Considering the PDG as a program representation (independent of the process that created it), the goal of PDG-reconstitution is to find a program whose PDG representation is the given PDG, much in the same way that the goal of an unparser (given an abstract syntax tree) is to find a program whose AST representation is the given AST. In this sense, the goal of PDG-reconstitution is merely syntactic.

However, an underlying premise of much of the work on dependence graphs is that if two programs have the same program dependence graph (*i.e.*, the graphs are isomorphic) then the two programs have the same meaning. Certainly, it would be troublesome if we took a program P, constructed its program dependence graph and reconstituted a program Q from this graph such that on some input for which P terminates and produces x, Q either does not terminate or does terminate and produces $y \neq x$. The equivalence property for program dependence graphs that correspond to programs written with only while loops, sequencing, and conditionals has been shown using a variety of frameworks [13, 15-17, 19]., However, to our knowledge, no one has extended the equivalence property for program dependence graphs with more complex control dependence. Such a property would provide a semantic justification for a reconstitution algorithm.

6.3. Maintaining Program Syntax

The CDG-reconstitution process described in this paper is independent of the source language and its syntax—the product of reconstitution is a control-flow graph rather than program text. In some applications that manipulate PDGs, such as program integration [14], it is desirable for the output program to resemble the input programs (when possible). In these cases, it is useful to annotate the PDG with information about the source program (syntax) from which it was derived. These annotations can be used by reconstitution to recreate the desired syntactic structure. We are examining the relationship between a language's syntax and PDGs, and are developing techniques for maintaining program syntax in environments that manipulate PDG representations of programs.

REFERENCES

1. A. Aho, J.E. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, MA (1974).

2. A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques and Tools,* Addison-Wesley, Reading, MA (1986).

3. B. Baker, "An Algorithm for Structuring Flow Graphs," *J. ACM* **24**(1) pp. 98-120 New York, NY, (January 1977).

4. T. Ball, S. Horwitz, and T. Reps, "Correctness of an algorithm for reconstituting a program from a dependence graph," Technical Report #947, Computer Sciences Department, University of Wisconsin – Madison, Madison, WI (July 1990).

5. W. Baxter and H. R. Bauer, III, "The Program Dependence Graph and Vectorization," *Proceedings of the Sixteenth ACM Principles of Programming Languages Symposium,* pp. 1-11 (January 11-13, 1989).

6. R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and K. Zadeck, "An efficient method of computing static single assignment form," pp. 25-35 in *Conference Record of the Sixteenth ACM Symposium on Principles of Programming Languages,* (Austin, TX, Jan. 11-13, 1989), ACM, New York, NY (1989).

7. 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)* **25**(6)(June 20-22, 1990).

8. D.E. Denning and P.J. Denning, “Certification of programs for secure information flow,” *Commun. of the ACM* **20**(7) pp. 504-513 (July 1977).

9. J. Ferrante and M. Mace, “On linearizing parallel code,” pp. 179-189 in *Conference Record of the Twelfth ACM Symposium on Principles of Programming Languages*, (New Orleans, LA, January 14-16, 1985), ACM, New York, NY (1985).

10. J. Ferrante, K. Ottenstein, and J. Warren, “The program dependence graph and its use in optimization,” *ACM Transactions on Programming Languages and Systems* **9**(5) pp. 319-349 (July 1987).

11. J. Ferrante, M. Mace, and B. Simons, “Generating Sequential Code From Parallel Code,” *Proceedings of the ACM 1988 International Conference on Supercomputing*, pp. 582-592 (July 1988).

12. D. Harel and R. E. Tarjan, “Fast Algorithms for Finding Nearest Common Ancestors,” *SIAM Journal of Computing* **13**(2)Society for Industrial and Applied Mathematics, (May 1984).

13. S. Horwitz, J. Prins, and T. Reps, “On the adequacy of program dependence graphs for representing programs,” pp. 146-157 in *Conference Record of the 15th ACM Symposium on Principles of Programming Languages,* (San Diego, CA, January 13-15, 1988), ACM, New York (1988).

14. S. Horwitz, J. Prins, and T. Reps, “Integrating non-interfering versions of programs,” *ACM Trans. Program. Lang. Syst.* **11**(3) pp. 345-387 (July 1989).

15. G. Ramalingam and T. Reps, “Semantics of Program Representation Graphs,” Technical Report #900, University of Wisconsin, Madison (December 1989).

16. T. Reps and W. Yang, “The semantics of program slicing,” Technical Report #777, Computer Sciences Department, University of Wisconsin, Madison, WI (June 1988).

17. T. Reps and W. Yang, “The semantics of program slicing and program integration,” in *Proceedings of the Colloquium on Current Issues in Programming Languages,* (Barcelona, Spain, March 13-17, 1989), *Lecture Notes in Computer Science*, Springer-Verlag, New York, NY (March 1989).

18. T. Reps, “Algebraic Properties of Program Integration,” *Proceedings of the 3rd European Symposium on Programming (Copenhagen, Denmark, May 15-18, 1990), Lecture Notes in Computer Science* **432**Springer-Verlag, (1990).

19. R. P. Selke, “A Rewriting Semantics for Program Dependence Graphs,” *Conference Record of the Sixteenth ACM Symposium on Principles of Programming Languages* (Austin, TX, Jan. 11-13, 1989), pp. 12-24 ACM, (1989).

20. B. Simons, B. Alpern, and J. Ferrante, “A Foundation for Sequentializing Parallel Code,” *Proceedings of the Symposium on Parallel Algorithms and Architectures*, (July 1990).

21. R. Tarjan, “Testing Flow Graph Reducibility,” *Proceedings of the 5th ACM Symposium on Theory of Computing*, pp. 96-107 (1973).

22. W. Yang, S. Horwitz, and T. Reps, “A program integration algorithm that accommodates semantics-preserving transformations,” in *Proceedings of the Fourth Symposium on Software Development Environments (to appear)*, ACM, New York, NY (December 1990).

Glossary of Terminology

(a,b) pair
: A distinct pair of vertices (a and b) with a common control parent in a CDG; *i.e.*, $\exists$ vertex p and label L such that $p \longrightarrow_c^L a$ and $p \longrightarrow_c^L b$.

backedge
: Edge $v \rightarrow w$ is a *backedge* of a reducible graph iff w **dom** v.

BF(C)
: Given reducible CDG C, the *backedge free* graph BF(C) is the graph obtained by removing all backedges from C.

control dependence
: $x \longrightarrow_c^L y$ (in CDG(G)) iff in CFG G w postdominates the L-branch of v and w does not postdominate v.

corresponding CFG
: G is a *corresponding CFG* of CDG C iff CDG(G) is isomorphic to C (the isomorphism map must respect vertex text and edge labels).

domination
: v **dom** w (in a CFG or a CDG) iff every path from *ENTRY* to w contains v (every vertex dominates itself).

DomReach(C)
: $\{ v \mid \forall x \in$ Reachable(C,v): v **dom** x in C $\}$.

feasible
: C is a *feasible* CDG iff it has a corresponding CFG

good order
: *Order* is a *good order* for CDG C if there is a corresponding CFG G such that for all (a,b) pairs in C, *Order* specifies that a **pd** b iff a **pd** b in G.

over
: b **over** a (by edge $v \longrightarrow_c^L b$) in CDG C iff there exists $v \longrightarrow_c^L b$ such that (not b **dom** v in C) and (not $v \longrightarrow_c^L a$).

parent(a)-free path
: A path in CDG C is *parent(a)-free* iff for all edges $v \longrightarrow_c^L w$ in the path, $v \longrightarrow_c^L a$ is not in C.

parent(a,b)-free path
: A path in CDG C is parent(a,b)-free iff for all edges $x \longrightarrow_c^L y$ in the path, not ($x \longrightarrow_c^L a$ and $x \longrightarrow_c^L b$) in C.

postdomination
: w **pd** v (in CFG G) iff $w \neq v$ and w is on every path from v to the *EXIT* vertex.

: Vertex w *postdominates* the L-branch of predicate vertex v iff w is the L-successor of v or w postdominates the L-successor of v.

: w **impd** v (in CFG G) iff w **pd** v and there is no vertex x such that w **pd** x **pd** v.

: pd(y) = $\{ x \mid x$ **pd** y $\}$.

: imm-pd(G,v) = w such that w **impd** v in CFG G.

Reachable(C,v)
: $\{ x \mid v \longrightarrow_c^* x$ in CDG C $\}$.

reducibility
: A graph (rooted at *ENTRY*) is *reducible* iff for each backedge $v \rightarrow w$ of G (as defined by a depth-first search of G starting at the *ENTRY* vertex) w **dom** v.

R-feasible
: C is an *R-feasible* CDG iff it has a corresponding CFG in the restricted class of CFGs (see Section 2.2).

APPENDIX

This appendix proves the results stated previously in this paper. Section 1 proves the most general results, properties *OrderFixed* and *OrderArbitrary*. Section 2 proves that *OrderFixed* and *OrderArbitrary* are complete for loop-free feasible CDGs (the CDGs of loop-free CFGs) and that *OrderFixed* is equivalent to *OrderFixed′* for this class of CDGs. The next three sections prove two main results concerning cyclic CDGs and *R*-feasible CDGs: Section 3 proves that the CDGs of reducible CFGS are reducible; Section 4 describes the restricted class of CFGs (those that give rise to the *R*-feasible CDGs) and some important properties of these CFGs. Section 5 shows that removing the backedges from an *R*-feasible CDG (which must be reducible) yields a feasible CDG. Using the results from the previous sections, Section 6 proves that the properties *OrderFixed* and *OrderArbitrary* are complete and that *OrderFixed* is equivalent to *OrderFixed′* for (possibly cyclic) *R*-feasible CDGs. Section 7 proves the correctness of property *Permutations*. Finally, Section 8 proves the correctness of function ConstructCFG and the isomorphism test.

1. General Results

The proofs of properties *OrderFixed* and *OrderArbitrary* rely on some basic results about control dependence presented in Section 1.1. Properties *OrderFixed* and *OrderArbitrary* are proved in Sections 1.2 and 1.3, respectively.

1.1. Basic Results About Control Dependence

LEMMA(1.1). If CFG G has vertices v, w, and z such that z **pd** v and $v \longrightarrow_c w$ is in CDG(G), then there is a non-empty z-free path from v to w in G.

PROOF. By contradiction. Since $v \longrightarrow_c w$ there must be a non-empty path from v to w. Suppose that every such path contains z. The facts that not w **pd** v and z **pd** v imply that not w **pd** z. Because z is on every path from v to w and not w **pd** z, w cannot postdominate the L-branch of v, as required by $v \longrightarrow_c^L w$. Contradiction. □

LEMMA(1.2). If CFG G has vertices v, w, and z such that z **pd** v and $v \longrightarrow_c w$ in CDG(G), then z **pd** w.

PROOF. By lemma (1.2), there must be a non-empty z-free path from v to w. If there were a z-free path from w to *EXIT* then z could not postdominate v. Therefore, z **pd** w. □

LEMMA(1.3). If CFG G has vertices a, b, and x such that b **pd** a and $a \longrightarrow_c^+ x$ in CDG(G), then b **pd** x.

PROOF. Let $a \longrightarrow_c x_1 \longrightarrow_c \cdots \longrightarrow_c x_n$ be a control dependence path from a to x where $x_n = x$. By lemma (1.2), b **pd** a and $a \longrightarrow_c x_1$ implies b **pd** x_1. Given b **pd** x_i and $x_i \longrightarrow_c x_{i+1}$, $1 \le i < n$, lemma (1.2) implies that b **pd** x_{i+1}. Therefore, b **pd** x. □

LEMMA(1.4). $v \longrightarrow_c^+ w$ in CDG(G) iff there is a non-empty path from v to w in G that does not contain any postdominators of v.

PROOF.

($\Rightarrow$) Let $v \longrightarrow_c x_1 \longrightarrow_c \cdots \longrightarrow_c x_n$ be a control dependence path from v to w where $x_n = w$. By lemma (1.3), any postdominator of v must postdominate all x_i. Since $v \longrightarrow_c x_1$, there must be a non-empty path from v to x_1. By lemma (1.1), one of these paths must be pd(v)-free. By similar reasoning, for all i ($1 \le i < n$), since $x_i \longrightarrow_c x_{i+1}$ and any postdominator of v postdominates x_i, there must be a non-empty, pd(v)-free path from x_i to x_{i+1}. Therefore, there must be a non-empty, pd(v)-free path from v to w.

($\Leftarrow$) Proof by induction on the number of predicates in the pd(v)-free path *PTH* from v to w (not including w). If there is one predicate in *PTH*, it must be v (if v were a statement vertex *PTH* would not be pd(v)-free). Since every other vertex in *PTH* is a statement, w must postdominate a branch of v. Since not w **pd** v, it follows that $v \longrightarrow_c w$. The induction hypothesis is that if there are fewer than N predicates in *PTH*, then $v \longrightarrow_c^+ w$. The induction step follows: since *PTH* is pd(v)-free, the induction hypothesis implies that all the predicates in *PTH* (except v and w) are transitively control dependent on v. Let p be the last predicate in *PTH* such that not w **pd** p (such a p must exist since not w **pd** v). It is clear that w postdominates every statement and predicate vertex between p and w in *PTH*. Therefore, w postdominates a branch of p and $p \longrightarrow_c w$ exists. Since $v \longrightarrow_c^+ p$, it follows that $v \longrightarrow_c^+ w$. □

LEMMA(1.5). If a **pd** v in CFG G (where a is not the *EXIT* vertex), then every path from *ENTRY* to v in CDG(G) must include an edge $p \longrightarrow_c^L x$ such that $p \longrightarrow_c^L a$ and a **pd** x in G (*i.e.*, there is no path from *ENTRY* to v that is parent(a)-free).

PROOF. By induction on the length of the path *PTH* from *ENTRY* to v in CDG(G).

Base Case: length of *PTH* = 1. In this case, $PTH = ENTRY \longrightarrow_c^T v$ and $p = ENTRY$ and $x = v$. Since $a \neq EXIT$, it follows that not a **pd** *ENTRY*. Furthermore, since a **pd** v and v postdominates the T-branch of *ENTRY*, it follows that a postdominates the T-branch of *ENTRY*. Therefore, $ENTRY \longrightarrow_c^T a$ exists. Since $x = v$ and a **pd** v, it follows that a **pd** x.

Induction Hypothesis: If a **pd** v then for all *PTH* of length $< N$, there is an edge $p \longrightarrow_c^L x$ in *PTH* such that $p \longrightarrow_c^L a$ and a **pd** x.

Induction Step: *PTH* of length $= N$. Suppose the control dependence parent of v in *PTH* ($p \longrightarrow_c^L v$) is not postdominated by a. Since a **pd** v it follows that $p \longrightarrow_c^L a$. Instead, suppose a **pd** p. $p \longrightarrow_c^L a$ obviously cannot exist. Let PTH_p be the prefix of *PTH* up to predicate p. The length of this path is less than N. Therefore, by the induction hypothesis, since a **pd** p, there must be a $q \longrightarrow_c^{L'} x$ in PTH_p such that $q \longrightarrow_c^{L'} a$ and a **pd** x. □.

1.2. Property *OrderFixed*

PROPERTY *OrderFixed*. Consider an (a,b) pair in feasible CDG C. If there is a parent(a)-free path *PTH* from *ENTRY* to a vertex in Reachable(C,b), then b **pd** a in every corresponding CFG of C.

PROOF. By contradiction. Assume all conditions, except that a **pd** b in some G (since a and b share a common parent, either b **pd** a or a **pd** b must hold). Let v be the vertex in Reachable(C,b) to which there is a parent(a)-free path from *ENTRY*. By lemma (1.3), $b \longrightarrow_c^* v$ and a **pd** b imply that a **pd** v. Lemma (1.5) implies that *PTH* is not parent(a)-free, a contradiction. □

1.3. Property *OrderArbitrary*

PROPERTY *OrderArbitrary*. Consider an (a,b) pair in feasible CDG C. If not a **over** b, not b **over** a, and $a,b \in$ DomReach(C), then there is a corresponding CFG in which b **pd** a, and a corresponding CFG in which a **pd** b.

The correctness of property *OrderArbitrary* follows immediately from the following lemma, which is used in the proof of property *Permutations*.

LEMMA (1.6). Consider an (a,b) pair in feasible CDG C and a corresponding CFG in which b **pd** x **pd** a (or b **impd** a). If not a **over** b, not b **over** a, and $a,b \in$ DomReach(C), then there is a corresponding CFG in which a **pd** x **pd** b (a **impd** b).

PROOF. Consider the special case where a and b are statement vertices. Since a and b have no outgoing edges in C, neither dominates any vertex besides itself in C. This fact together with (not a **over** b) and (not b **over** a) implies that the sets $\{ (v,L) \mid v \longrightarrow_c^L a \}$ and $\{ (v,L) \mid v \longrightarrow_c^L b \}$ are equal. Thus, if G is a corresponding CFG in which b **pd** x **pd** a (or b **impd** a), then simply renaming a to b and vice-versa yields a corresponding CFG G' in which a **pd** x **pd** b (a **impd** b).

To extend this proof to the case where a and b are predicates requires the notion of a control-flow hammock. A hammock is a single-entry, single-exit subgraph of the CFG (it is useful to think of a hammock as a “super vertex”). When the conditions of property *OrderArbitrary* hold, it is possible to swap the hammocks with entries a and b in a corresponding CFG in which b **pd** a to yield a corresponding CFG in which a **pd** b. The remainder of this section proves this in greater detail.

DEFINITION: A *hammock* H is an induced subgraph of CFG G with entry vertex v in H and exit vertex v' not in H such that:

1. All edges from $(G-H)$ to H go to v.
2. All edges from H to $(G-H)$ go to v'.

DEFINITION: Let imm-pd(G,x) denote the immediate postdominator of vertex x in CFG G.

LEMMA(1.7). Consider a vertex v in CDG C with corresponding CFG G. Let $H =$ Reachable(C,v). Vertex v dominates all vertices in H in C iff H is a hammock in G where v is the entry and imm-pd(G,v) is the exit.

PROOF.

($\Leftarrow$) By lemma (1.4), every vertex in H must be transitively control dependent on v (because no vertex in H can postdominate v). Since H is a hammock, all edges from $G-H$ to H go to v. This implies for any vertex $w \in H$ and $w \neq v$, w cannot postdominate the L-branch of a vertex outside of H. Therefore, w cannot be directly control dependent on a vertex outside of H, and v dominates all vertices in H in C.

($\Rightarrow$) We show that the two conditions for H (Reachable(C,v)) to be a hammock must hold:

(1) Suppose that there is an edge $x \rightarrow y$ in CFG G where $x \notin$ Reachable(C,v), $y \in$ Reachable(C,v), and $v \neq y$. If not y **pd** x in G, then $x \longrightarrow_c y$, which contradicts the assumption that v dominates Reachable(C,v). Therefore, y **pd** x in G. By lemma (1.5), along any path from *ENTRY* to x in C there must be an edge $p \longrightarrow_c^L z$ in the path such that $p \longrightarrow_c^L y$. If $p \notin$ Reachable(C,v) then v cannot dominate Reachable(C,v). If $p \in$ Reachable(C,v), then since $p \longrightarrow_c^+ x$, it follows that $x \in$ Reachable(C,v). Contradiction.

(2) Suppose that there is an edge $x \rightarrow y$ in CFG G where $x \in V(H)$, $y \in V(G-H)$, and $y \neq$ imm-pd(G,v). Since $v \longrightarrow_c^+ x$, lemma (1.4) implies that there is a pd(v)-free path from v to x in G. If not y **pd** v then there is a pd(v)-free path from v to y and lemma (1.4) implies that $v \longrightarrow_c^+ y$. This implies that $y \in$ Reachable(C,v), a contradiction of $y \in V(G-H)$. Therefore, y **pd** v. However, since $v \longrightarrow_c^+ x$, there must be a pd(v)-free path from v to x in G. Therefore, there is a path from v to y in G in which y is the first postdominator of v in the path. This implies that $y =$ imm-pd(G,v). Contradiction. □

We are now in a position to prove lemma (1.6) in its full generality. We assume that the following hold for the given (a,b) pair: not a **over** b, not b **over** a, and $a,b \in$ DomReach(C). We assume, without loss of generality, that C has a corresponding CFG G in which b **pd** x **pd** a (or b **impd** a). We must show that C also has a corresponding CFG G' in which a **pd** x **pd** b (a **impd** b).

Since both a and b are in DomReach(C), lemma (1.7) implies that G must contain a hammock H_a = Reachable(C,a) with entry a and exit imm-pd(G,a), and a hammock H_b = Reachable(C,b) with entry b and exit imm-pd(G,b). Hammocks are either disjoint or nested. Since b **pd** a in CFG G and no vertex in H_a can postdominate a, H_a and H_b must be disjoint.

The CFG G' contains the same vertex set as G. The edge set of CFG G' is defined as follows:

$$
\begin{aligned}
& \{\, v \to^L w \mid v \to^L w \text{ is in } G \text{ and } [\, (v,w \notin H_a \cup H_b) \text{ or } (v,w \in H_a) \text{ or } (v,w \in H_b) \,] \,\} \\
\cup\ & \{\, v \to^L b \mid v \to^L a \text{ is in } G \text{ and } v \notin H_a \,\} \\
\cup\ & \{\, v \to^L a \mid v \to^L b \text{ is in } G \text{ and } v \notin H_b \,\} \\
\cup\ & \{\, v \to^L \text{imm-pd}(G,b) \mid v \to^L \text{imm-pd}(G,a) \text{ is in } G \text{ and } v \in H_a \,\} \\
\cup\ & \{\, v \to^L \text{imm-pd}(G,a) \mid v \to^L \text{imm-pd}(G,b) \text{ is in } G \text{ and } v \in H_b \,\}
\end{aligned}
$$

The relationship between CFGs G and G' is depicted in Figure 6. It is obvious that a **pd** x **pd** b (or a **impd** b) in CFG G' and that H_a and H_b are hammocks in CFG G'. To show that G' corresponds to C we must show that CDG(G') is identical to C. This follows from the four observations below (the last three have symmetrical counterparts replacing a with b and H_a with H_b):

(1) Since hammocks are single-entry and single-exit subgraphs, swapping hammocks H_a and H_b cannot affect any control dependence between vertices v and w that are both outside of $(H_a \cup H_b)$.

(2) Since swapping the hammocks does not change any paths strictly inside H_a and does not change any postdomination ordering between vertices strictly inside H_a, there is no effect on control dependence between vertices v and w that are both inside H_a.

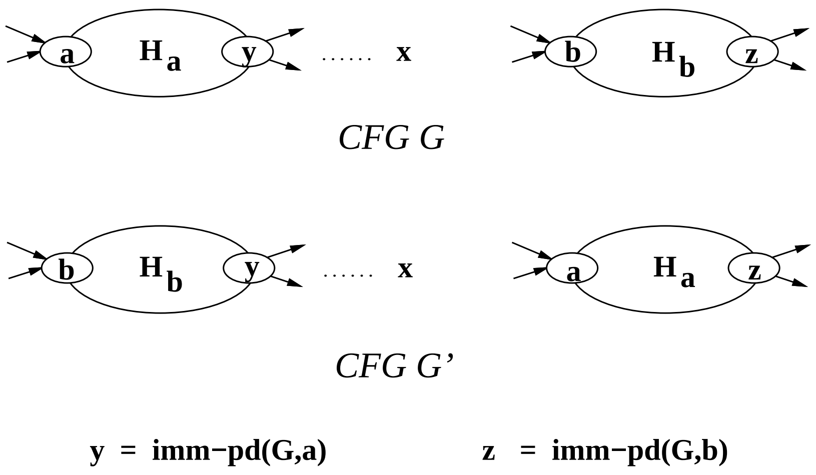


Figure 6.

(3) Since (not a **over** b) and (not b **over** a) in CDG C, swapping the hammocks cannot affect the control dependence predecessors of a that lie outside of H_a.

(4) Since H_a is a hammock in both G and G', lemma (1.7) implies that there can be no control dependence in either C or CDG(G') from a vertex inside H_a to a vertex outside of H_a, or from a vertex outside of H_a to a vertex inside of H_a other than a. □

2. Results for Loop-Free Feasible CDGs

This section shows that properties *OrderFixed* and *OrderArbitrary* are complete for loop-free feasible CDGs and that property *OrderFixed′* is equivalent to property *OrderFixed* for this class of CDGs. These results rely on the following two lemmas about CDGs and loops:

LEMMA (2.1). There is a loop in CFG G iff there is a loop in CDG(G).

PROOF. Straightforward, from the definition of control dependence. □

LEMMA (2.2). Consider an (a,b) pair in CDG C of CFG G. If there is a vertex x such that $x \in$ Reachable(C,a) and $x \in$ Reachable(C,b), then there is a loop in C.

PROOF. Without loss of generality, assume that b **pd** a in G. This implies that $x \neq b$ (if $x = b$ then $a \longrightarrow_c^+ b$, which is not possible since b **pd** a). Since $x \in$ Reachable(C,a) and b **pd** a, lemma (1.3) implies that b **pd** x. There must be a path in G from b to x (because $b \longrightarrow_c^+ x$) and a path in G from x to b (because b **pd** x); therefore, b and x participate in a loop in G. By lemma (2.1), there must be a loop in CDG C. □

2.1. Completeness of Ordering Properties

In this section we show that either property *OrderFixed* or property *OrderArbitrary* holds for every (a,b) pair in a loop-free feasible CDG (it is clearly impossible for *OrderFixed* to hold in both directions in a feasible CDG). The proof relies on the following definition and lemma:

DEFINITION. A path in a CDG is parent(a,b)-free iff for each edge $x \longrightarrow_c^L y$ in the path, not ($x \longrightarrow_c^L a$ and $x \longrightarrow_c^L b$).

LEMMA (2.3). Consider an (a,b) pair in CDG C. If there is no parent(a)-free path (from *ENTRY*) to a vertex in Reachable(C,b), and no parent(b)-free path to a vertex in Reachable(C,a), then for any z in Reachable(C,a) $\cup$ Reachable(C,b), there is no parent(a,b)-free path from *ENTRY* to z.

PROOF. By contradiction. Assume the antecedent but that there is a z in Reachable(C,a) $\cup$ Reachable(C,b) such that there is a parent(a,b)-free path *PTH* from *ENTRY* to z. Consider the parent(a)-free and parent(b)-free prefix (possibly empty) of *PTH*. Let $p \longrightarrow_c^L x$ be the edge in *PTH* immediately after the prefix. Either $p \longrightarrow_c^L a$ or $p \longrightarrow_c^L b$ (but not both) exists. This implies that there is parent(b)-free path to a or a parent(a)-free path to b (consisting of the prefix followed by the edge $p \longrightarrow_c^L a$ or $p \longrightarrow_c^L b$). Contradiction. □

LEMMA (2.4). Properties *OrderFixed* and *OrderArbitrary* are complete for loop-free feasible CDGs.

PROOF. By contradiction. We suppose that there is an (a,b) pair in a loop-free feasible CDG C for which neither property holds and yield the contradiction that C must contain a loop. If property *OrderFixed* does not hold in either direction for an (a,b) pair, then there is no parent(a)-free path (from *ENTRY*) to a vertex in Reachable(C,b), nor is there a parent(b)-free path to a vertex in Reachable(C,a). Lemma (2.3) implies that for any z in Reachable(C,a) $\cup$ Reachable(C,b), there is no parent(a,b)-free path from *ENTRY* to z. Suppose that property *OrderArbitrary* does not hold, either because (1) $a \notin$ DomReach(C) or (2) a **over** b:

(1) Suppose that $a \notin$ DomReach(C) because there is some $v \notin$ Reachable(C,a) and $w \in$ Reachable(C,a) such that $v \longrightarrow_c w$ and $w \neq a$. Since there can be no parent(a,b)-free path from *ENTRY* to w, on every path from *ENTRY* to w that ends with $v \longrightarrow_c w$ there must be an edge $x \longrightarrow_c^L y$ such that $x \longrightarrow_c^L a$ and $x \longrightarrow_c^L b$. If $x = v$ then $y = w$ and $y \neq a$ (because $w \neq a$). If $x \neq v$ then $y \neq a$ (otherwise $v \in$ Reachable(C,a)). In either case, vertices y and a are distinct, share a common control parent (x) and reach a common vertex (w), so lemma (2.2) implies that there must be a loop in C. Contradiction.

(2) Suppose that a **over** b by edge $v \longrightarrow_c^L a$. Since there can be no parent(a,b)-free path from *ENTRY* to a, on every path from *ENTRY* to v there must be an edge $x \longrightarrow_c^L y$ such that $x \longrightarrow_c^L a$ and $x \longrightarrow_c^L b$. Since a **over** b by $v \longrightarrow_c^L a$, $v \neq x$, which implies that $y \neq a$. Vertices y and a are distinct, share a common control parent (x) and reach a common vertex (a), yielding the same contradiction as in (1). □

2.2. Equivalence of properties *OrderFixed* and *OrderFixed′*

LEMMA (2.5). If CDG C is loop-free and feasible, then property *OrderFixed* implies that b **pd** a iff property *OrderFixed′* implies that b **pd** a.

PROOF.

($\Rightarrow$) If C is loop-free and feasible and property *OrderFixed* implies that b **pd** a, then property *OrderFixed′* implies that b **pd** a. Since property *OrderFixed* implies that b **pd** a, there must be a parent(a)-free path from *ENTRY* to a vertex in Reachable(C,b). This implies that either $b \notin$ DomReach(C) or b **over** a. If not a **over** b, then either case (a) or (b) of property *OrderFixed′* must imply that b **pd** a.

Instead, suppose that a **over** b by edge $z \longrightarrow_c^{L'} a$. We show that case (c) of property *OrderFixed′* implies that b **pd** a. By the definition of a **over** b, $z \longrightarrow_c^{L'} b$ cannot be in C. Let *PTH* be a path from *ENTRY* to a ending with edge $z \longrightarrow_c^{L'} a$. Since b **pd** a in every corresponding CFG of C, lemma (1.5) implies that there must be an edge $v \longrightarrow_c^L x$ in *PTH* such that $v \longrightarrow_c^L b$ and b **pd** x in every corresponding CFG of C. It is clear that $v \longrightarrow_c^L x \longrightarrow_c^+ a$. To show that b **over** a by edge $v \longrightarrow_c^L b$, we must show that (not b **dom** v) and not $v \longrightarrow_c^L a$. If b **dom** v then since $v \longrightarrow_c^L b$, it follows that there is a loop in the CDG. Suppose instead that $v \longrightarrow_c^L a$. If $x = a$ then $z \longrightarrow_c^{L'} a = v \longrightarrow_c^L x$, which implies that $z \longrightarrow_c^{L'} b$ exists (because of $v \longrightarrow_c^L b$), a contradiction. If $x \neq a$ then since $x \longrightarrow_c^+ a$ and x and a have a common parent (v), lemma (2.2) implies that there is a loop in the CDG.

($\Leftarrow$) If C is loop-free and feasible and property *OrderFixed′* implies that b **pd** a, then property *OrderFixed* implies that b **pd** a. Assume that property *OrderFixed′* implies that b **pd** a but that property *OrderFixed* does not imply that b **pd** a. Since case (a), (b), or (c) of property *OrderFixed′* must hold, either $b \notin$ DomReach(C) or b **over** a, which implies that property *OrderArbitrary* cannot hold.

Since C is feasible, property *OrderFixed* does not imply that b **pd** a and property *OrderArbitrary* does not hold, it follows that property *OrderFixed* implies that a **pd** b. By ($\Rightarrow$), property *OrderFixed′* must imply that a **pd** b. We will show a contradiction in each of the three possible cases:

(1) Case (a) implies that a **pd** b. Note that neither (b) nor (c) can imply that b **pd** a. Suppose that (a) implies that b **pd** a: $b \notin$ DomReach(C) because of edge $v \longrightarrow_c w$ (where $v \notin$ Reachable(C,b), $w \in$ Reachable(C,b), and $w \neq b$). Since a **pd** b in all corresponding CFGs of C and w is transitively control dependent on b, lemma (1.3) implies that a **pd** w. By lemma (1.5), since a **pd** w, on any path from *ENTRY* to w there must be an edge $p \longrightarrow_c^L x$ such that $p \longrightarrow_c^L a$. Since not a **over** b, either a **dom** p or $p \longrightarrow_c^L b$. Since $p \longrightarrow_c^L a$, if the former case holds then there is a loop in CDG C. Instead, suppose that $p \longrightarrow_c^L b$. Since b and x reach a common vertex w and have a common control parent, there must be a loop in CDG C.

(2) Case (b) implies that a **pd** b. It is not possible for (a), (b), or (c) to imply that b **pd** a (because (b) and (c) both require b **over** a, and (a) requires not a **over** b). Therefore, *OrderFixed′* cannot imply that b **pd** a, as assumed.

(3) Case (c) implies that a **pd** b. Neither (a) nor (b) can imply that b **pd** a. Suppose that (c) implies that b **pd** a: a **over** b and b **over** a by edge $v \longrightarrow_c^L b$ such that $v \longrightarrow_c^L x \longrightarrow_c^+ a$. Since a **pd** b in all corresponding CFGs of C, $v \longrightarrow_c^L b$, and not $v \longrightarrow_c^L a$, it follows that a **pd** v (if not a **pd** v then $v \longrightarrow_c^L a$ would exist). Therefore, by lemma (1.4), a cannot be transitively control dependent on v. □

3. The CDGs of Reducible CFGs are Reducible

This section shows that the CDG of a reducible CFG is itself reducible. There are many equivalent definitions of reducibility. A graph G (rooted at *ENTRY*) is reducible iff any of the following is true:

(1) For each back edge $v \rightarrow w$ of G (as identified by a depth-first search of G starting at the *ENTRY* vertex) w **dom** v.

(2) Every depth-first search of G identifies the same set of edges as backedges.

(3) There is no loop in G that contains distinct vertices a and b such that (not a **dom** b) and (not b **dom** a) in G.

The following lemma relates domination in the CFG to domination in the CDG:

LEMMA (3.1). If v dominates w in reducible CFG G and $v \longrightarrow_c^* w$ CDG(G), then v dominates w in CDG(G).

PROOF. Suppose that v dominates w in reducible CFG G and that $v \longrightarrow_c^* w$ CDG(G). Let H be the set of vertices that are dominated by v and postdominated by imm-pd(G,v) in G. Suppose there is a v-free path in CDG(G) from *ENTRY* to w. Since $ENTRY \notin H$ and $w \in H$, this path must include an edge $z \longrightarrow_c z'$ such that $z \notin H$ and $z' \in H$. We will show that no vertex in H other than v can be directly control dependent on a vertex z outside H; thus, this path cannot exist and v dominates w in CDG(G). There are two cases to consider, depending on whether or not there is a v-free control flow path from z to a vertex in H:

(1) Every control flow path from z to a vertex u in H contains v. If $u = v$ then u can be directly control dependent on z. However, if $u \neq v$, then u cannot be directly control dependent on z, as we now argue. Since every control flow path from z to u must pass through v and no vertex in H can postdominate v, it is impossible for u to postdominate the T-branch (or F-branch) of z. Thus, u could not be control dependent on z.

(2) There is a v-free control flow path *PTH* from z to a vertex in H. Without loss of generality assume that the last vertex (u) in *PTH* is in H and that all other vertices in *PTH* are not in H. Let $t \rightarrow u$ be the last edge in *PTH*. We show that u cannot dominate t in G and that there is a depth-first search of G that identifies $t \rightarrow u$ as a backedge. This implies that G is irreducible, yielding a contradiction.

It should be clear that no vertex in H (except v) can dominate a vertex outside of H. All that remains to be shown is that there is a cycle-free path from *ENTRY* to t that includes u (which implies that some depth-first search would identify $t \rightarrow u$ as a backedge). There is a cycle-free path PTH_0 from *ENTRY* to v such that no vertex in PTH_0 (except v) is in H. There is a cycle-free path PTH_1 from v to imm-pd(G,v) that includes u such that every vertex in PTH_1 (except imm-pd(G,v)) is in H. There is a cycle-free path PTH_2 from imm-pd(G,v) to t such that no vertex in PTH_2 is in H. The only way a cycle could arise in the concatenation of the three paths is if PTH_2 and PTH_0 share a vertex x ($x \neq v$, since no vertex in PTH_2 is in H). Concatenating the prefix of PTH_0 up to and including x, the suffix of prefix of PTH_2 starting at x, and the edge $t \rightarrow u$, yields a v-free path from *ENTRY* to u, contradicting

u's membership in H. Therefore ($PTH_0 \parallel PTH_1 \parallel PTH_2$) must be cycle-free.□

LEMMA (3.2). If CFG G is reducible then CDG(G) is reducible.

PROOF. Let C = CDG(G). Suppose G is reducible and that C is irreducible. We will show a contradiction: namely, that G must be irreducible. Since C is irreducible, there must be a backedge $v \longrightarrow_c w$ in C such that w does not dominate v. Furthermore, v cannot dominate w in C (because the depth-first search that identifies $v \longrightarrow_c w$ as a backedge must encounter w before v). The following points result:

(1) Since w is directly control dependent on v in C, there must be a control flow path from v to w in G.

(2) By lemma (3.1), since v does not dominate w in C, $v \longrightarrow_c w$, and G is reducible, v cannot dominate w in G. Therefore, there must be a v-free control flow path from *ENTRY* to w in G.

(3) Since $v \longrightarrow_c w$ is a backedge, there must be a chain of control dependences from w to v. Therefore, there must be a control flow path from w to v in G.

(4) It is obvious that there must be a control flow path from *ENTRY* to v. We must show that there is such a path that is w-free. Suppose the contrary. If w dominates v in G, then since G is reducible and v is transitively control dependent on w, w must dominate v in C by lemma (3.1). This contradicts an initial assumption.

By points (1) through (4), we have shown that there is a loop in G containing distinct vertices v and w such that not w **dom** v and not v **dom** w in G. Therefore, G is irreducible. □

4. The Restricted Class of CFGs

The R-feasible CDGs are defined as the CDGs of a *restricted* class of CFGs. We use the concept of a natural loop in a CFG to define this class of CFGs.

DEFINITION. The natural loop of a backedge $v \rightarrow w$ is defined as nat-loop($v \rightarrow w$) = $\{ w \} \cup \{ x \mid$ there is a w-free path from x to $v \}$.

DEFINITION. The natural loop of a loop entry w, denoted nat-loop(w), is the union of all nat-loop($v \rightarrow w$), where $v \rightarrow w$ is a backedge.

A CFG G is in the *restricted class* iff it is reducible and for each loop entry w (a vertex that is the target of one or more backedges), no vertex in nat-loop(w) postdominates w.

There are several important properties concerning natural loops in reducible CFGs:

(1) A loop entry w dominates all vertices in nat-loop(w). This follows immediately from reducibility.

(2) Given loop entry w, there is a path from w to each vertex in nat-loop(w) that contains only vertices in nat-loop(w). This follows immediately from (1).

(3) Given two loop entries w and x, from the definition of natural loop it follows that either nat-loop(w) and nat-loop(x) are disjoint, or one contains the other. This allows us to identify levels of loop nesting and outermost loops (*i.e.*, loops that are not contained in any other loops).

5. The Feasibility of Backedge-free R-feasible CDGs

Given a reducible CDG C, its backedge-free counterpart (BF(C)) is uniquely defined. While BF(C) is a loop-free CDG, it is not necessarily a feasible CDG. For example, removing the self-looping backedge from the feasible (but R-infeasible) CDG in Figure 4(b) yields an infeasible CDG. The main result of this section is that for any R-feasible CDG C, BF(C) is feasible.

LEMMA(5.1). If CDG C is R-feasible, then BF(C) is feasible.

PROOF. Given a CFG G in the restricted class, we show that BF(CDG(G)) is feasible. The proof is by induction on the number of backedges in G.

Base Case: If G has no backedges then, by lemma (2.1), CDG(G) has no backedges either. This implies that BF(CDG(G)) = CDG(G).

Induction Hypothesis: If G has fewer than N backedges ($N > 0$) then BF(CDG(G)) is feasible.

Induction Step: If CFG G has N backedges ($N > 0$) then BF(CDG(G)) is feasible. The proof consists of three steps:

(1) Find an *outermost* loop in CFG G headed by loop entry w and pick a backedge $v \rightarrow w$. Construct CFG G' from G as follows: G' = (vertices(G), edges(G) − { $v \rightarrow w$ } ∪ { $v \rightarrow$ imm-pd(G,w) }).

(2) Show that G' is in the restricted class and contains fewer than N backedges. This implies that CDG(G') is reducible, so BF(CDG(G')) is uniquely defined.

(3) Show that BF(CDG(G)) = BF(CDG(G')).

Since G' is in the restricted class and contains fewer than N backedges, the Induction Hypothesis implies that BF(CDG(G')) is feasible. Since BF(CDG(G)) = BF(CDG(G')), BF(CDG(G)) is feasible.

We now turn to proving (2) and (3). The proofs of these steps rely on the following properties relating postdomination and domination in G and G'. In the remainder of this section, z denotes the vertex imm-pd(G,w). The proofs make use of the following two observations: any $v \rightarrow w$-free path in G is also in G'; any $v \rightarrow z$-free path in G' is also in G.

(a) $\forall x,\ \forall y$: not y **pd** x in $G \Rightarrow$ not y **pd** x in G'.

If there is a y-free path from x to *EXIT* in G that is $v \rightarrow w$-free, then the same path is clearly in G'. If the only y-free path from x to *EXIT* in G contains $v \rightarrow w$, then there must be a y-free and $v \rightarrow w$-free path from x to v and from w to *EXIT* (and from z to *EXIT*, since z **pd** w) in G. The same paths are in G'. Since v is connected to z by $v \rightarrow z$ in G', it follows that not y **pd** x in G'.

(b) $\forall y, y \neq w$: not y **pd** x in $G' \Rightarrow$ not y **pd** x in G.

If there is a y-free path from x to *EXIT* in G' that is $v \rightarrow z$-free, then the same path is in G. If the only y-free path *PTH* from x to *EXIT* in G' contains $v \rightarrow z$, then there must be a y-free and $v \rightarrow z$-free path from x to v and from z to *EXIT* in G'. Thus, the same paths are in G. If there is a y-free path from w to z in G, then not y **pd** x in G. If y is on every path from w to z in G, then since z = imm-pd(G,w), it follows that either $y = w$ or $y = z$. The first case contradicts the assumption that $y \neq w$. The second case implies that *PTH* is not y-free.

(c) $\forall x, x \notin$ nat-loop(w) in G : not w **pd** x in $G' \Rightarrow$ not w **pd** x in G.

If there is a w-free path from x to *EXIT* in G' that is $v \rightarrow z$-free, then the same path is in G. If the only w-free path *PTH* from x to *EXIT* in G' contains $v \rightarrow z$, then there must be a w-free and $v \rightarrow z$-free path from x to v in G', and thus in G. However, since $x \notin$ nat-loop(w) and $v \in$ nat-loop(w), any path from x to v in G must contain w.

(d) $\forall x,\ \forall y$: not y **dom** x in $G \Rightarrow$ not y **dom** x in G'.

If there is a y-free path from *ENTRY* to x in G that is $v \rightarrow w$-free, then the same path is in G'. It is not possible that the only y-free path from *ENTRY* to x in G contains $v \rightarrow w$, since w **dom** v in G.

(e) $\forall x,\ \forall y$: not y **dom** x in $G' \Rightarrow$ not y **dom** x in G.

If there is a y-free path from *ENTRY* to x in G' that is $v \rightarrow z$-free, then the same path is in G. If the only y-free path *PTH* from *ENTRY* to x in G' contains $v \rightarrow z$, then there is a y-free path *PTH′* (a prefix of *PTH*) from *ENTRY* to v and a y-free path from z to x in G. Since w **dom** v in G, w occurs in *PTH′* and thus in *PTH*. If there is a y-free path from w to z in G, then there is a y-free path from *ENTRY* to x in G. If y is on every path from w to z in G, then since z = imm-pd(G,w), either $y = z$ or $y = w$. In the former case, *PTH* cannot be y-free. In the latter case, since $y = w$ and w is in *PTH*, *PTH* cannot be y-free.

- **G' is reducible**.

Since z = imm-pd(G,w) and G is in the restricted class, z cannot be in nat-loop(w). This and the fact that nat-loop(w) is an outermost loop imply that w and z cannot participate in a loop in G. Because z = imm-pd(G,w), there can be no path from z to w in G. It follows that there can be no path from z to a member of nat-loop(w) in G. Thus, the edge $v \rightarrow z$ cannot participate in any loop in G'.

A graph is irreducible iff it contains a cycle with distinct vertices a and b such that not a **dom** b and not b **dom** a. Suppose that such a situation occurs in G'. Since $v \rightarrow z$ cannot participate in this loop, the same loop exists in G. Furthermore, (e) implies that not a **dom** b and not b **dom** a in G. This implies that G is irreducible.

- **G' contains fewer than N backedges**

Since G and G' are reducible and y **dom** x in G iff y **dom** x in G' (by (d) and (e)), it follows that any backedge in G (excluding $v \rightarrow w$) is a backedge in G', and that any backedge in G' (with the exception of $v \rightarrow z$) is a backedge in G. We show that $v \rightarrow z$ cannot be a backedge in G' (not z **dom** v in G'), which implies that G' contains $N-1$ backedges. As shown above, z and v cannot participate in a loop in G. Since there is a path from v to z in G (as $v \rightarrow w$ and z = imm-pd(G,w)), it follows that not z **dom** v in G. Thus, (d) implies that not z **dom** v in G'.

- **G' is in the restricted class**.

We prove that for each loop entry d in G', no vertex in nat-loop(d) postdominates d. As shown above, every backedge in G' is also a backedge in G. Since $v \rightarrow z$ cannot participate in any loop, for each backedge $c \rightarrow d$ in G', nat-loop($c \rightarrow d$) in G' is equal to nat-loop($c \rightarrow d$) in G. Consider any vertex f in nat-loop($c \rightarrow d$) in G'. Vertex f is also in nat-loop($c \rightarrow d$) in G. Since G is in the restricted class it follows that not f **pd** d. By (a), not f **pd** d in G'. Thus, G' is in the restricted class.

- **BF(CDG(G)) = BF(CDG(G'))**.

We prove that BF(CDG(G)) = BF(CDG(G')) by showing the following relationships between CDG(G) and CDG(G'):

(i) $\forall x,\ \forall y, y \neq w : x \longrightarrow_c^L y$ in CDG(G) $\Leftrightarrow x \longrightarrow_c^L y$ in CDG(G').

From (a) and (b) it follows that if $y \neq w$ then not y **pd** x in G iff not y **pd** x in G', and that y postdominates the L-branch of x in G iff y postdominates the L-branch of x in G'. Therefore, if $y \neq w$, then $x \longrightarrow_c^L y$ in CDG(G) iff $x \longrightarrow_c^L y$ in CDG(G').

(ii) $\forall x, x \notin$ nat-loop(w) in $G : x \longrightarrow_c^L w$ in CDG(G) $\Leftrightarrow x \longrightarrow_c^L w$ in CDG(G').

From (a) and (c) it follows that if $x \notin$ nat-loop(w), then not w **pd** x in G iff not w **pd** x in G', and that w postdominates the L-branch of x in G iff w postdominates the L-branch of x in G'.

(iii) $\forall x,\ x \in$ nat-loop(w) in G : w **dom** x in CDG(G) and CDG(G'). This implies that if $x \longrightarrow_c^L w$ is in CDG(G) or CDG(G'), then it is a backedge of that CDG.

Since $x \in$ nat-loop(w) in G, w **dom** x in G. By (e), w **dom** x in G'.

Since G is reducible and no member of nat-loop(w) can postdominate w in G, there is a path *PTH* from w to x in G containing no postdominators of w (without loss of generality, we can assume that *PTH* is $v \rightarrow w$ free, since inclusion of $v \rightarrow w$ would imply that *PTH* contains a cycle). Since *PTH* is $v \rightarrow w$-free in G, *PTH* exists in G'. Since no vertex in *PTH* postdominates w in G, (a) implies that no vertex in *PTH* postdominates w in G'. Thus, lemma (1.4) implies that $w \longrightarrow_c^+ x$ in CDG(G) and in CDG(G').

Since w **dom** x in G and G', and $w \longrightarrow_c^+ x$ in CDG(G) and CDG(G'), lemma (3.1) implies that w **dom** x in CDG(G) and in CDG(G').

(iv) $\forall x,\ \forall y : y$ **dom** x in CDG(G) $\Leftrightarrow$ y **dom** x in CDG(G').

Follows from (i), (ii), and (iii). We prove ($\Leftarrow$). The proof of ($\Rightarrow$) is symmetrical (switch CDG(G) and CDG(G')). Suppose that not y **dom** x in CDG(G). Let *PTH* be a y-free path from *ENTRY* to x in CDG(G). Note that we can always pick *PTH* such that for each edge $a \longrightarrow_c b$ in *PTH*, not b **dom** a in CDG(G). *PTH* cannot contain an edge of the form $a \longrightarrow_c w$, where $a \in$ nat-loop(w) in G, because (iii) implies that w **dom** a in CDG(G). Thus, (i) and (ii) imply that *PTH* is in CDG(G') and not y **dom** x in CDG(G').

The above four points imply that the only edges on which CDG(G) and CDG(G') can differ are backedges (edges for which the target dominates the source in the CDG). Thus, BF(CDG(G)) = BF(CDG(G')). □

6. Results for *R*-feasible CDGs

In this section we show that properties *OrderFixed* and *OrderArbitrary* are complete for *R*-feasible CDGs, and that properties *OrderFixed* and *OrderFixed′* are equivalent for *R*-feasible CDGs. The following lemmas characterize some important relationships between *R*-feasible CDG C and BF(C).

LEMMA(6.1). If CDG C is *R*-feasible, then

(1) b **over** a in BF(C) iff b **over** a in C, and

(2) $b \notin$ DomReach(BF(C)) $\Rightarrow$ $b \notin$ DomReach(C)

PROOF.

(1) ($\Leftarrow$) If b **over** a by $v \longrightarrow_c^L b$ in C, then $v \longrightarrow_c^L b$, not b **dom** v, and not $v \longrightarrow_c^L a$. Since C is reducible and not b **dom** v, it follows that $v \longrightarrow_c^L b$ cannot be a backedge. Thus, b **over** a by $v \longrightarrow_c^L b$ in BF(C).

($\Rightarrow$) Suppose b **over** a in BF(C) by $v \longrightarrow_c^L b$ (*i.e.*, $v \longrightarrow_c a$ is not in BF(C) and not b **dom** v in BF(C)). Since C is reducible, it follows that y **dom** x in C iff y **dom** x in BF(C). Therefore, not b **dom** v in C. Suppose that $v \longrightarrow_c^L a$ is in C. Since $v \longrightarrow_c^L a$ is a backedge it follows that a **dom** v in C, and thus in BF(C). This means that $a \longrightarrow_c^+ b$ in BF(C). Lemma (2.2) implies that there is a loop in BF(C). Contradiction.

(2) If $b \notin$ DomReach(BF(C)), then there must be a vertex x ($x \neq b$) in Reachable(BF(C),b) such that there is a b-free path from *ENTRY* to x in BF(C). The same path must obviously exist in C.

LEMMA(6.2). Given an (a,b) pair in *R*-feasible CDG C. If $b \in$ DomReach(BF(C)) and $b \notin$ DomReach(C), then there is a parent(a)-free path from *ENTRY* to a member of Reachable(C,b).

PROOF. Since C is reducible, b **dom** x in BF(C) iff b **dom** x in C. If no vertex in Reachable(BF(C),b) is the source of a backedge in C, then $b \in$ DomReach(C). If for every vertex v in Reachable(BF(C),b) that is the source of a backedge $v \longrightarrow_c w$ in C, w is in Reachable(BF(C),b), then $b \in$ DomReach(C).

The only case left is where there is a vertex v in Reachable(BF(C),b) that is the source of a backedge $v \longrightarrow_c w$ in C, and $w \notin$ Reachable(BF(C),b). Since C is reducible, w must dominate b in this case. We show that w **dom** a, which implies that there must be a parent(a)-free path to w (a member of Reachable(C,b)).

Let p be the common parent of vertices a and b. Since w **dom** b and $p \longrightarrow_c b$, either $w = p$ or w **dom** p. It follows that $w \longrightarrow_c^+ a$ and that $b \longrightarrow_c^+ a$. Since a and b have a common parent, either b **pd** a or a **pd** b in a corresponding CFG. However, by lemma (1.4), if a **pd** b in a corresponding CFG then $b \longrightarrow_c^+ a$ cannot exist. Therefore, b **pd** a in all corresponding CFGs. Suppose that not w **dom** a by some path *PTH*. Since b **pd** a in all corresponding CFGs, lemma (1.5) implies that there is some $x \longrightarrow_c^L y$ in *PTH* such that $x \longrightarrow_c^L b$. This implies that there is an w-free path from *ENTRY* to b consisting of the prefix of *PTH* up to x, followed by the edge $x \longrightarrow_c^L b$. This contradicts the fact that w **dom** b. □

LEMMA(6.3). If C is R-feasible and there is a parent(a)-free path from *ENTRY* to a vertex in Reachable(C,b) in BF(C), then the same path is parent(a)-free in C.

PROOF. By contradiction. If the path is not parent(a)-free in C, then it must be because of a backedge ($v \longrightarrow_c a$). Since C is reducible, it follows that a **dom** v. This implies that a is in the parent(a)-free path in BF(C), which is clearly a contradiction as a parent(a)-free path cannot contain a. □

6.1. Completeness of Properties *OrderFixed* and *OrderArbitrary*

PROPERTY *Complete*. Properties *OrderFixed* and *OrderArbitrary* are complete for R-feasible CDGs.

PROOF. To show that properties *OrderFixed* and *OrderArbitrary* are complete for a given R-feasible CDG C, we consider which property holds in BF(C). Since BF(C) is feasible and loop-free, one of the properties must hold for a given (a,b) pair.

(A) Assume that property *OrderFixed* holds (in some direction) in BF(C) for (a,b). Lemma (6.3) implies that property *OrderFixed* holds in the same direction in C.

(B) Assume that property *OrderArbitrary* holds in BF(C) for (a,b). If property *OrderArbitrary* holds in C, then we are done. If it does not hold, then lemma (6.1) part (1) implies that either a or b is not a member of DomReach(C). Lemma (6.2) implies that property *OrderFixed* must hold in C. □

6.2. Equivalence of Properties *OrderFixed* and *OrderFixed′*

LEMMA (6.4). If CDG C is R-feasible, then property *OrderFixed* implies that b **pd** a iff property *Order-Fixed′* implies that b **pd** a.

PROOF.

($\Rightarrow$) Assume that property *OrderFixed* implies that b **pd** a. We will show that property *OrderFixed′* implies that b **pd** a, by case analysis on which property (*OrderArbitrary* or *OrderFixed*) holds in BF(C).

If property *OrderArbitrary* holds in BF(C), then by lemma (6.1) part (1) either a or b is not a member of DomReach(C). If $a \notin$ DomReach(C), then by lemma (6.2) property *OrderFixed* must imply a **pd** b, contradicting an initial assumption. Therefore, $b \notin$ DomReach(C). Since property *OrderArbitrary* holds in BF(C), by lemma (6.1) part (1), not (a **over** b) and not (b **over** a) in C. Thus, case (a) of property

OrderFixed′ holds.

If property *OrderFixed* holds in BF(C), then lemma (6.3) implies that it must hold in the same direction as in C. Therefore, property *OrderFixed′* implies that b **pd** a in BF(C). By lemma (6.1), it is clear that property *OrderFixed′* also must imply that b **pd** a in C.

($\Leftarrow$) Assume that property *OrderFixed′* implies that b **pd** a. We will show that property *OrderFixed* implies that b **pd** a.

By lemma (6.1), it is clear that if case (b) or (c) of property *OrderFixed′* implies that b **pd** a in C, then the same case implies that b **pd** a in BF(C). Furthermore, if case (a) of property *OrderFixed′* implies that b **pd** a and $b \notin$ DomReach(BF(C)), then case (a) implies that b **pd** a in BF(C). Since BF(C) is feasible, property *OrderFixed* implies that b **pd** a in BF(C). By lemma (6.3), it follows that *OrderFixed* implies that b **pd** a in C.

If case (a) implies that b **pd** a in C and $b \in$ DomReach(BF(C)), then, by lemma (6.2), property *OrderFixed* implies that b **pd** a in C. □

7. Property *Permutations*

The proof of property *Permutations* relies on the following lemma.

LEMMA (7.1). Given feasible CDG C. If property *OrderArbitrary* holds for (a, b) and b **pd** x **pd** a in a corresponding CFG of C, then (1) there is a vertex p such that $p \longrightarrow_c^L a$, $p \longrightarrow_c^L x$, and $p \longrightarrow_c^L b$, and (2) property *OrderArbitrary* holds for (a,x) and (x,b).

PROOF. There must be a vertex p such that $p \longrightarrow_c^L a$ and $p \longrightarrow_c^L b$. Since b **pd** x **pd** a, it follows that $p \longrightarrow_c^L x$. By lemma (1.6), there must be a corresponding CFG G' in which a **pd** x **pd** b, which implies (2). □

PROPERTY *Permutations*. If CDG C is R-feasible, then an order for the (a,b) pairs of C is a good order iff it (1) respects the fixed pair orderings determined by property *OrderFixed* and (2) orders the (a,b) pairs for which property *OrderArbitary* holds according to an arbitrary total ordering of C's vertices.

PROOF.

($\Rightarrow$) If *Order* is a good order, then *Order* must be acyclic and respect the fixed orderings of property *OrderFixed*. Since *Order* is acyclic it must be possible to find a total ordering of C's vertices that respects the pair orderings of those (a,b) pairs for which property *OrderArbitrary* holds.

($\Leftarrow$) Since property *OrderArbitrary* cannot hold for an (a,b) pair that spans regions (see Section 3.1.3) and regions partition the vertex set of C, we only need to argue that within any region the (a,b) pairs for which property *OrderArbitrary* holds can be ordered according to an arbitrary order of the vertices in that region. Within any region $R = \{ x_1,...,x_n \}$ of an R-feasible CDG C, at most one x_k cannot be a member of DomReach(C). Such an x_k must postdominate all other vertices in the region. Property *OrderArbitrary* holds for every pair of vertices from the set $S = (R - \{ x_k \mid x_k \notin \text{DomReach}(C) \})$.

Consider a corresponding CFG of C, and let x_n be the vertex from S such that x_n postdominates all other vertices in S and let x_1 be the vertex from S such that all other vertices in S postdominate x_1. This chain contains all the vertices in S and no others (if there was an $x_k \notin S$ such that x_n **pd** x_k **pd** x_1 then lemma (7.1) implies that property *OrderArbitrary* holds for (x_n,x_k) and (x_k,x_1) which implies that $x_k \in S$). Since property *OrderArbitrary* holds for every pair in the chain, lemma (1.6) implies that there is a corresponding CFG for every possible postdomination ordering of vertices in S. □

8. The Correctness of Function ConstructCFG

In this section we assume that CDG C is feasible and *Order* is a good order for C. We first prove two lemmas relating immediate postdominators in a corresponding CFG to CDG C and *Order*. These results are used to show that function ConstructCFG builds a corresponding CFG (when given a feasible CDG and a good order). Finally, we show that the CDG of this CFG is isomorphic to C via a mapping constructed by uniquely tagging the vertices of C and carrying the tags over to the vertices of G and CDG(G). In what follows, let the list *CLIST*(p, L) = ($v_1,...,v_m$) be the L-children of p ordered so that for all i, $1 \le i < m$, the ordered pair (v_i, v_{i+1}) is in *Order* (*i.e.*, *Order* specifies that v_{i+1} **pd** v_i).

LEMMA (8.1). Let G be a corresponding CFG of C that respects *Order* (*i.e.*, the graphs representing *Order* and the good order of G are isomorphic under an isomorphism map between C and CDG(G)). If w immediately follows v in CLIST(p,L), then w = imm-pd(G,v).
PROOF. By contradiction. Suppose x ($\neq w$) is the immediate postdominator of v in G. Since x **impd** v and w **pd** v in G (since w occurs after v in CLIST(p,L)), w **pd** x **pd** v in G. Since $p \longrightarrow_c^L w$ and $p \longrightarrow_c^L v$, and w **pd** x **pd** v, it follows that $p \longrightarrow_c^L x$ is in C. Therefore, x must occur between v and w in CLIST(p,L). Contradiction. □

LEMMA (8.2). Let G be a corresponding CFG of C that respects *Order*. If vertex v occurs last in CLIST(q,L) and r = imm-pd(G,q), then r = imm-pd(G,v).

PROOF. Assume that that $w \neq r$ is the immediate postdominator of v in G. Since r **impd** q in G and v is directly control dependent on q, lemma (1.2) implies that r **pd** v in G. Since w **pd** v and r **pd** v in G, either w **pd** r or r **pd** w must hold in G. The former implies that w is not the immediate postdominator of v in G. Therefore, r **pd** w **pd** v. Since v postdominates the L-branch of q and w **pd** v, but not w **pd** q (since r **pd** w), $q \longrightarrow_c^L w$ must be in C. However, since w **pd** v, w must follow v in CLIST(q,L), contradicting an initial assumption. □

LEMMA (8.3). Let G be a corresponding CFG of C that respects *Order*. Function ConstructCFG builds the CFG G (since ConstructCFG is deterministic this implies that G is the only corresponding CFG of C that respects *Order*).
PROOF. We will show that function ConstructCFG correctly determines the edge set of G. To do so we first show that for any call to DFS (see Figure 3) with parameters v and w, w = imm-pd(G,v). The first call to DFS is DFS(*ENTRY*,*EXIT*). The vertex *EXIT* is the immediate postdominator of *ENTRY* in every CFG. Suppose the call DFS(v,w) has been made and that w = imm-pd(G,v). A call to DFS made by this invocation of DFS either has the form DFS(x,y), where y immediately follows x in a CLIST of v, or the form DFS(x,w), where x is the last vertex in a CLIST of v. In the first case, lemma (8.1) implies that y = imm-pd(G,x). In the second case, lemma (8.2) implies that w = imm-pd(G,x).

In any CFG, the control-flow successor of a statement vertex is its immediate postdominator. Since function ConstructCFG correctly determines the immediate postdominator of each vertex in G, it is clear that it correctly determines the control-flow successors of statement vertices. We now show that function ConstructCFG correctly determines the *true* and *false* control-flow successors of each predicate p in G. There are two cases to consider for a given predicate p:

1. p has no L-successors in C. In this case, ConstructCFG determines that imm-pd(G,p) is the L-successor of p in G. Suppose that x ($x \neq$ imm-pd(G,p)) is the L-successor of p in G. This implies that $p \longrightarrow_c^L x$ is in C, which contradicts the fact that p has no L-successors in C.

2. p has L-successors in C. In this case, ConstructCFG determines that the first vertex in CLIST(p,L) is the L-successor of p in G. Let x be the L-successor of p in G. Vertex x cannot postdominate p, otherwise there would be no L-successors of p in C. This implies that $p \longrightarrow_c^L x$. Since x is the L-successor of p in G, it must be the first vertex in CLIST(p,L) (because every other vertex that is L control dependent on p must postdominate x). □

LEMMA (8.4). Let G be the corresponding CFG of C that respects *Order*, as constructed by function ConstructCFG. CDG(G) is isomorphic to C via a mapping constructed by uniquely tagging the vertices of C and carrying the tags over to the vertices of G and CDG(G).

PROOF. Suppose that no two vertices in C contain the same text. In this case, there is only one possible isomorphism map between CDG(G) (which contains the same vertex set as C) and C. The tagging procedure clearly computes this map. However, the text in a vertex plays no part in function ConstructCFG or in the computation of CDG(G) from G. Therefore, even if some vertices in C contain the same text, the tagging procedure will compute an isomorphism map between C and CDG(G). □