# The Multi-Procedure Equivalence Theorem

DAVID BINKLEY, SUSAN HORWITZ, and THOMAS REPS
University of Wisconsin — Madison

Program dependence graphs have been used in program optimization, vectorization, and parallelization. They have also been used as the internal representation for programs in programming environments, as well as for automatically merging program variants.

This paper concerns the question of whether program dependence graphs are "adequate" as a program representation. Previous results on the adequacy of program dependence graphs have been limited to a language without procedures and procedure calls. Our main result is a theorem that extends previous results to a language with procedures and procedure calls. The theorem shows that if the program dependence graphs of two programs are isomorphic then the programs are strongly equivalent.

CR Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs — *control structures, procedures, functions, and subroutines;* D.3.4 [**Programming Languages**]: Processors — *compilers, optimization;* E.1 [**Data**]: Data Structures — *graphs, trees*

General Terms: Theory

Additional Key Words and Phrases: activation tree, aliasing, control dependence, control-flow graph, data dependence, data-flow analysis, program dependence graph, strong equivalence, system dependence graph

## 1. INTRODUCTION

Dependence graph representations of programs have been used in program optimization, vectorization, and parallelization [9[13], [10]11], as the internal representation for programs in a language-based programming environment [12], and for automatic program integration [6]. The semantic equivalence of programs with isomorphic program dependence graphs was first demonstrated in [7], the main result of which was the proof of the following theorem:

THEOREM. (EQUIVALENCE THEOREM). *If P and Q are programs with isomorphic program dependence graphs, then for any initial state σ, either both P and Q diverge when initiated on σ, or both halt with the same final state.*

Restated in the contrapositive the theorem reads: Inequivalent programs have non-isomorphic program dependence graphs.

This work was supported in part by a David and Lucile Packard Fellowship for Science and Engineering, by the National Science Foundation under grants DCR-8552602 and 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 IBM, DEC, Xerox, and Kodak.

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

The language treated by Horwitz et al. contains scalar variables, assignment statements, conditional statements, and while-loops. This paper extends the Equivalence Theorem of [7] to the *Multi-Procedure Equivalence Theorem*, which applies to programs that include procedures and procedure calls. To distinguish between the more limited language of [7] and the language treated in this paper, we call programs with procedures and procedure calls "*systems*". The graphs used to represent systems are called *system dependence graphs*. We prove the semantic equivalence of systems with isomorphic system dependence graphs by reducing this question to the question of the semantic equivalence of *programs* with isomorphic *program* dependence graphs.

The proof proceeds as follows: Given systems $S$ and $T$ with isomorphic system dependence graphs, when $S$ is initiated on an arbitrary initial state $\sigma$, either $S$ diverges or $S$ terminates. In the case that $S$ diverges, we show that $T$ also diverges. In the case that $S$ terminates, we show that $S$ and $T$ can be transformed into programs $\hat{S}$ and $\hat{T}$ (with no procedures or procedure calls) such that when initiated on state $\sigma$, $\hat{S}$ and $\hat{T}$ produce the same final states as $S$ and $T$, respectively, and $\hat{S}$ and $\hat{T}$ have isomorphic program dependence graphs. By the Equivalence Theorem, $\hat{S}$ and $\hat{T}$ are semantically equivalent, from which it follows that systems $S$ and $T$ are also semantically equivalent.

The proof relies on two lemmas, the Expansion Lemma and the Flattening Lemma. The Expansion Lemma demonstrates that, given systems $S$ and $T$ with isomorphic system dependence graphs, "in-line" expansion of corresponding procedure call statements produces systems $S'$ and $T'$ such that (1) $S'$ is semantically equivalent to $S$, (2) $T'$ is semantically equivalent to $T$, and (3) the system dependence graphs of $S'$ and $T'$ are isomorphic. For a language without recursion, the Expansion Lemma alone is sufficient to show the existence of programs $\hat{S}$ and $\hat{T}$ as described above, so that the Multi-Procedure Equivalence Theorem follows from the Equivalence Theorem.

To prove the Multi-Procedure Equivalence Theorem for a language *with* recursion, we consider an arbitrary initial state $\sigma$ on which execution of system $S$ terminates. Because $S$ terminates, the number of procedure calls made during its execution is finite. The Flattening Lemma proves the existence of a "flattening" operation such that Flatten($S$, $\sigma$) is a system that, when initiated on state $\sigma$, makes *no* procedure calls and produces the same final state as does system $S$ when initiated on $\sigma$. The flattening operation involves a sequence of call-statement expansions that remove executed call statements. The Flattening Lemma also proves that applying this sequence of expansions to system $T$ produces a system, $\bar{T}$, such that $\bar{T}$ and Flatten($S$, $\sigma$) have isomorphic system dependence graphs. The final step in the proof of the Multi-Procedure Equivalence Theorem is to show that it is possible to transform systems Flatten($S$, $\sigma$) and $\bar{T}$ to programs $\hat{S}$ and $\hat{T}$, with the properties described above.

The remainder of the paper is organized as follows. Section 2 summarizes the definition of the system dependence graph given in [8]. Section 3 states and proves the Multi-Procedure Equivalence Theorem. Section 4 discusses related work.

## 2. SYSTEMS AND SYSTEM DEPENDENCE GRAPHS

A *system* consists of a main program and a set of auxiliary procedures (we assume that in a system all called procedures exist). Systems are written in a language with scalar variables, assignment statements, conditional statements, call statements, while loops, and a restricted kind of "output statement" called an *end statement*. An end statement, which can only appear at the end of the main program, lists zero or more variables; when execution terminates, only those variables will have values in the final state. Intuitively, the variables named by the end statement are those whose final values are of interest to the programmer.

We assume that all parameters are passed using call-by-reference parameter passing. The minor adaptations needed to handle call-by-value and call-by-value-result are straightforward. Call-by-reference parameter passing is the most interesting of the three because it introduces the possibility of aliasing. We also assume that systems contain no global variables; systems with global variables can be transformed into semantically equivalent systems without global variables by converting global variables into additional reference parameters.

### 2.1. The System Dependence Graph

This section summarizes the definition of system dependence graphs given in [8]. A system dependence graph includes a *program dependence graph*, which represents the system's main program, and a collection of *procedure dependence graphs*, which represent the system's auxiliary procedures (both program dependence graphs and procedure dependence graphs will be referred to as PDGs when the distinction between the two is irrelevant). PDGs are connected to form the system dependence graph by interprocedural dependence edges that represent dependences between a call site and the called procedure.

One of our goals in designing the system dependence graph is that the graph should consist of a straightforward connection of the program dependence graph and procedure dependence graphs. By "straightforward connection", we mean that the endpoints of interprocedural dependence edges should be the vertices associated with call sites and procedure entry, rather than *arbitrary* vertices in the calling and called procedures. To achieve this goal the system dependence graph models the following non-standard, two-stage mechanism for run-time parameter passing. When procedure *P* calls procedure *Q*, parameter addresses are transferred from *P* to *Q* by means of intermediate temporary variables. A different set of temporary variables is used when *Q* returns to transfer values back to *P*. Before the call, *P* copies the addresses of actual parameters into the call temporaries; procedure *Q* then initializes formal parameters from these temporaries. Occurrences of formal parameters in procedure *Q* are implicitly dereferenced. Before returning, *Q* copies return values into the return temporaries, from which *P* retrieves them.

The vertices used to represent a call statement in a PDG are determined from interprocedural summary information [1]. Two kinds of interprocedural summary information are determined for each procedure *P*:

$GMOD(P)$
: the set of formal parameters that might be *modified* by *P* itself or by a procedure (transitively) called from *P*.

$GREF(P)$
: the set of formal parameters that might be *referenced* by *P* itself or by a procedure (transitively) called from *P*.

The program dependence graph for the main program of system *S* contains the following vertices:

1) a distinguished *entry vertex* labeled "*Enter Main*";
2) an *initial-definition vertex* labeled "$x := InitialState(x)$", for each variable *x* in the main program of system *S* that may be referenced before being defined;
3) a *final-use vertex* labeled "FinalUse(x)" for each variable *x* appearing in the *end* statement;
4) a vertex representing each non-call statement and control predicate;
5) for each call statement of the form "call *P* (...)",
    a) a *call-site vertex* (which is considered a predicate vertex that always evaluates to **true** — see below),
    b) an *actual-in vertex* labeled "$f_{in} := AddrOf(p)$", for each actual parameter *p* corresponding to

formal parameter $f \in GMOD(P) \cup GREF(P)$[1], and

c) an *actual-out vertex* labeled “$p := f_{out}$”, for each actual parameter $p$ corresponding to formal parameter $f \in GMOD(P)$.

The procedure dependence graph for a procedure $P$ contains the same vertices as a program dependence graph except:

1) the entry vertex is labeled “*Enter P*”;
2) initial-definition vertices are labeled “$x := 0$”, where $x$ is a variable local to $P$;[2]
3) there are no final-use vertices;
4) there is a *formal-in vertex* labeled “$f := f_{in}$”, for each formal parameter $f \in GMOD(P) \cup GREF(P)$; and
5) there is a *formal-out vertex* labeled “$f_{out} := f$”, for each formal parameter $f \in GMOD(P)$.

The edges of a PDG represent *dependences* among procedure components.[3] An edge represents either a *control dependence* or a *data dependence*. The source of a control dependence edge is always the entry vertex, a call site vertex or a predicate vertex. A control dependence exists between $v_1$ and $v_2$, denoted by $v_1 \longrightarrow_c v_2$, if $v_2$ occurs on every control-flow graph path from $v_1$ to the end of the control-flow graph along one branch emanating from $v_1$ but not the other. A control dependence edge $v_1 \longrightarrow_c v_2$ is labeled by the truth value of the branch in which $v_2$ always occurs.

A method for determining control dependence edges for arbitrary programs is given in [4]; however, because we are assuming that programs include only assignment, conditional, call, and while statements, control dependence edges can be determined in a much simpler fashion. For the language under consideration here, procedure $P$’s PDG contains a *control dependence edge* from vertex $v_1$ to vertex $v_2$ iff one of the following holds:

1) $v_1$ is the entry vertex, and $v_2$ represents a component of $P$ that is not nested within any loop or conditional; these edges are labeled **true**.
2) $v_1$ is the entry vertex and $v_2$ represents either a formal-in, formal-out, initial-definition, or final-use vertex; these edges are labeled **true**.
3) $v_1$ is a call-site vertex and $v_2$ represents either an actual-in vertex or an actual-out vertex associated with the call statement; these edges are labeled **true**.
4) $v_1$ represents a control predicate, and $v_2$ represents a component of $P$ immediately nested within the loop or conditional whose predicate is represented by $v_1$. If $v_1$ is the predicate of a while-loop, the edge $v_1 \longrightarrow_c v_2$ is labeled **true**; if $v_1$ is the predicate of a conditional statement, the edge $v_1 \longrightarrow_c v_2$ is labeled **true** or **false** according to whether $v_2$ occurs in the **then** branch or the **else** breach, respectively.[4]

---

[1] By definition parameters in $GMOD(P)$ are only potentially modified. Thus for $x \in GMOD(P)$ procedure $P$ may return the initial value of $x$. For this reason, a PDG includes an actual-in vertex (part 5b) for each parameter in $GMOD(P)$.

[2] We assume that on entry to a procedure all the local variables of the procedure that may be referenced before being defined are initialized to zero.

[3] In the rest of this section, we use the term “procedure” as a generic term referring to both the main program and the auxiliary procedures.

[4] In other definitions that have been given for control dependence edges, there is an edge from the predicate of a **while** statement to itself labeled **true**. By including the additional edge, the predicate’s outgoing **true** edges consist of every program element that is guaranteed to be executed (eventually, assuming all loops and procedures calls terminate) when the predicate evaluates to **true**. Our control dependence edges are technically the *forward* control dependence edges defined in [3].

There is a data dependence edge from vertex $v_1$ to vertex $v_2$ if reversing the relative order of the components represented by $v_1$ and $v_2$ may alter the program's computation. In this paper, PDGs contain two kinds of data-dependence edges, representing *flow dependences* and *def-order dependences*. The data dependence edges of a PDG are computed from a control-flow graph representation of the procedure.[5]

A PDG contains a flow dependence edge from vertex $v_1$ to vertex $v_2$ (denoted by $v_1 \longrightarrow_f v_2$) iff all of the following hold:

1) $v_1$ is a vertex that defines variable $x$.
2) $v_2$ is a vertex that references variable $y$ or is an actual-in vertex that references the address of $y$.[6]
3) $x$ and $y$ are potential aliases [1,2].
4) Control can reach $v_2$ after $v_1$ via a path in the control-flow graph along which there is no intervening definition of $x$ or $y$.

Note that clause (4) does not exclude there being definitions of other variables that are potential aliases of $x$ or $y$ along the path from $v_1$ to $v_2$. An assignment to a variable $z$ along the path from $v_1$ to $v_2$ only over-writes the contents of the memory location written by $v_1$ if $x$ and $z$ refer to the same memory location. If $z$ is a potential alias of $x$, then there is only a *possibility* that $x$ and $z$ refer to the same memory location; thus, an assignment to $z$ does not necessarily over-write the memory location written by $v_1$, and it may be possible for $v_2$ to read a value written by $v_1$.

Flow dependences can be further classified as *loop-carried* or *loop-independent*. A flow dependence $v_1 \longrightarrow_f v_2$ is carried by loop $L$, denoted by $v_1 \longrightarrow_{lc(L)} v_2$, if in addition to 1), 2), 3), and 4) above, the following also hold:

5) There is a path in the control-flow graph that both satisfies the conditions of 4) above and includes a backedge to the predicate of loop $L$.
6) Both $v_1$ and $v_2$ are enclosed in loop $L$.

A flow dependence $v_1 \longrightarrow_f v_2$ is loop-independent, denoted by $v_1 \longrightarrow_{li} v_2$, if in addition to 1), 2), 3), and 4) above, there is a path in the control-flow graph that satisfies 4) above and includes *no* backedge to the predicate of a loop that encloses both $v_1$ and $v_2$. It is possible to have both $v_1 \longrightarrow_{lc(L)} v_2$ and $v_1 \longrightarrow_{li} v_2$.

A PDG contains a def-order dependence edge from vertex $v_1$ to vertex $v_2$ iff all of the following hold:

1) $v_1$ and $v_2$ define variables $x_1$ and $x_2$, respectively.
2) $x_1$ and $x_2$ are potential aliases.
3) $v_1$ and $v_2$ are in the same branch of any conditional statement that encloses both of them.
4) There exists a *witness vertex* $v_3$ such that $v_1 \longrightarrow_f v_3$ and $v_2 \longrightarrow_f v_3$.
5) $v_1$ occurs to the left of $v_2$ in the procedure's abstract syntax tree.

A def-order dependence edge from $v_1$ to $v_2$ with witness $v_3$ is denoted by $v_1 \longrightarrow_{do(v_3)} v_2$.

Note that a PDG is a multi-graph (*i.e.*, it may have more than one edge of a given kind between two vertices). When there is more than one loop-carried flow dependence edge between two vertices, each is labeled by a different loop that carries the dependence. When there is more than one def-order edge

---

[5] A description of the control-flow graphs used to compute data dependence edges appears in Section 2.2.

[6] Recall our goal of having the endpoints of interprocedural dependence edges be either vertices associated with a call site or vertices associated with procedure entry. By treating the actual-in vertex for parameter $y$ as a use of $y$, this vertex, rather than vertices that represent uses of the corresponding formal parameter, becomes the target for flow dependences from definitions of $x$ in the calling procedure.

between two vertices, each is labeled by a different witness vertex.

Connecting the PDGs to form the system dependence graph is straightforward, involving the addition of three kinds of interprocedural edges. *Call edges* represent the control dependence of a called procedure on the corresponding call site; *parameter-in* and *parameter-out* edges represent data dependences between formal and actual parameters. Interprocedural dependence edges connect the call site, actual-in, and actual-out vertices representing a call to procedure $P$ with the entry, formal-in, and formal-out vertices, respectively, of procedure $P$'s PDG. A system dependence graph has an interprocedural dependence edge from $v_1$ to $v_2$ iff one of the following holds:

1) $v_1$ is a call-site vertex and $v_2$ is the called procedure's entry vertex; this edge is a call edge.
2) $v_1$ is an actual-in vertex labeled "$f_{in} := AddrOf(p)$" and $v_2$ is a formal-in vertex labeled "$f := f_{in}$"; this edge is a parameter-in edge.
3) $v_1$ is a formal-out vertex labeled "$f_{out} := f$" and $v_2$ is an actual-out vertex labeled "$p := f_{out}$"; this edge is a parameter-out edge.

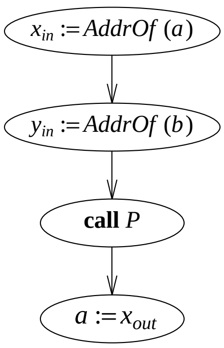
*Example*. Figure 1 shows a program to sum the numbers from 1 to $N$ and its system dependence graph.

## 2.2. Control-flow Graphs

Understanding the proof of the Multi-Procedural Equivalence Theorem (in particular the Path Blocking Lemma and the Expansion Lemma) requires having a more complete understanding of the control-flow graphs that represent the procedures of a system. A procedure's control-flow graph contains a unique start node, a node representing each assignment statement, call statement, if predicate, and while predicate in the procedure, and a sequence of assignment nodes representing each call statement's parameters. This sequence is composed of nodes that represent the copying of actual parameter addresses to call temporaries, followed by nodes that represent the assignment of return temporaries to actual parameters. The actual-in and actual-out vertices representing a call statement in a PDG correspond directly to the parameter nodes representing the call statement in the procedure's control-flow graph.

In addition, a procedure's control-flow graph starts with a sequence of formal-in nodes that copy values from call temporaries to formal parameters, and ends with a sequence of formal-out nodes that copy values from formal parameters to return temporaries. The formal-in and formal-out vertices in a PDG correspond directly to the formal-in and formal-out nodes, respectively, of a procedure's control-flow graph.

*Example*. Let procedure $P$ have two formal parameters $x$ and $y$. Assume that $GMOD(P) = \{ x \}$ and $GREF(P) = \{ x, y \}$ (recall that which values are transferred to and from a called procedure is determined from interprocedural summary information [1]). The statement "**call** $P(a, b)$" is represented by the four node sequence:



```
program                              procedure Add(x, y)
   sum := 0;                            x := x + y
   i := 1;                           return
   while i ≤ N do
        call Add(sum, i);
        call Add(i, 1)
   od
end(sum, i)
```

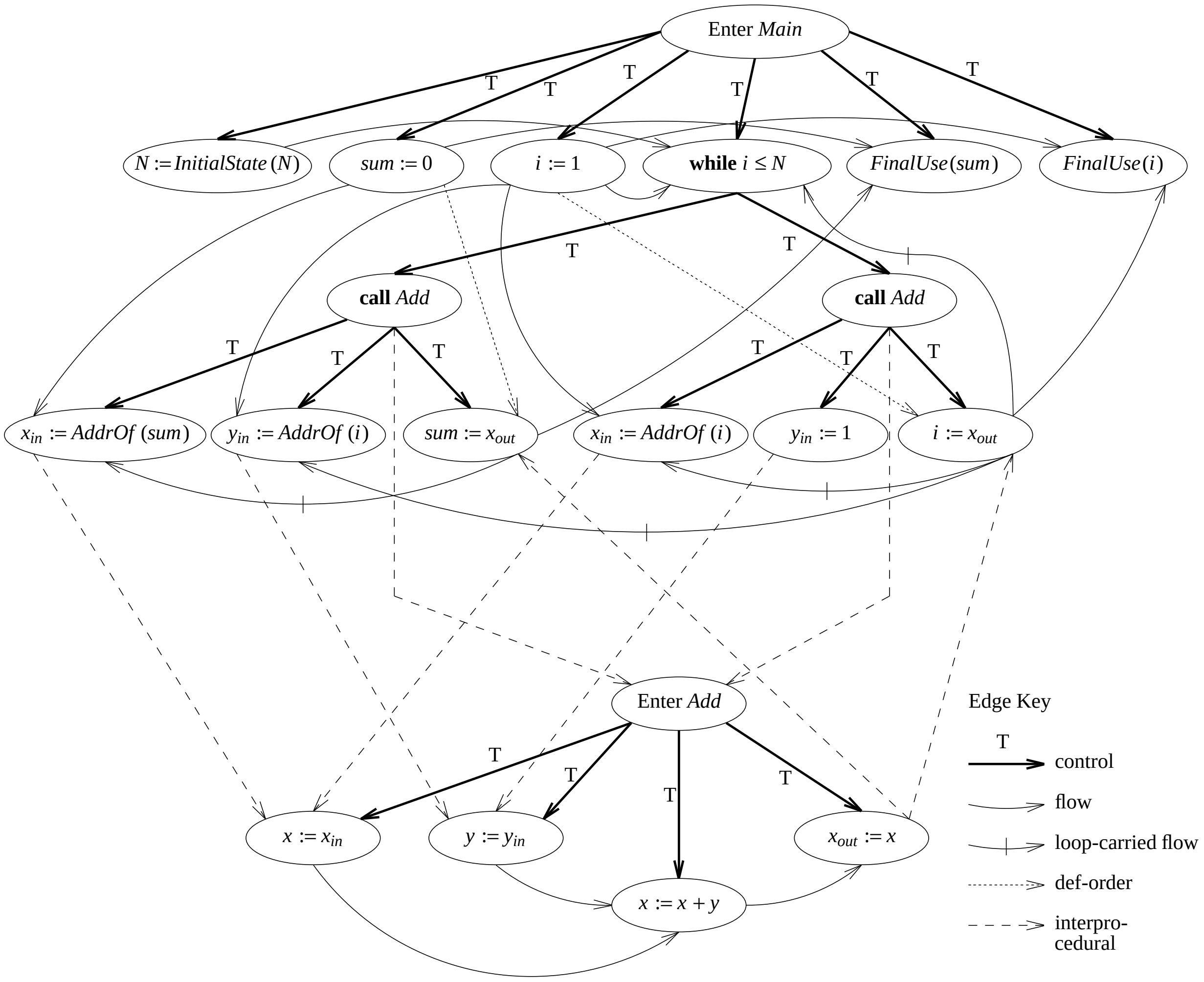



**Figure 1.** An example system and its system dependence graph. The boldface arrows represent control dependences, dotted arrows represent def-order dependences, solid arrows represent loop-independent flow dependences, solid arrows with a hash mark represent loop-carried flow dependences, and dashed arrows represent interprocedural dependences. Note that there is no formal-out vertex for parameter *y* in procedure *Add* because *y* is not in *GMOD*(*Add*).

The first two nodes are actual-in nodes; the last node is the only actual-out node.

## 3. THE MULTI-PROCEDURE EQUIVALENCE THEOREM

The following definitions are used in the statement and proof of the Multi-Procedure Equivalence Theorem.

*Definition*. Two systems $S$ and $T$ are *strongly equivalent* iff for any state $\sigma$, either $S$ and $T$ both diverge when initiated on $\sigma$ or they both halt with the same final state[7]. If $S$ and $T$ are not strongly equivalent, we say they are *inequivalent*.

*Definition*. A *region* is either the list of top level statements in a procedure or the list of statements immediately nested within a conditional statement or while-loop.

*Definition*. Two systems, $S$ and $T$ have *isomorphic* system dependence graphs (written $G_S \approx G_T$) iff the vertices of $G_S$ and $G_T$ have identical contents (labels) and there exists a 1-1 correspondence between the edge sets of $G_S$ and $G_T$.

When systems $S$ and $T$ have isomorphic system dependence graphs, the statements in each region in system $T$ are a permutation of the statements in the corresponding region in system $S$ such that the reordering in $T$ preserves the dependence edges in $G_S$.

Our principal result is the following theorem:

*THEOREM*. (MULTI-PROCEDURE EQUIVALENCE THEOREM). *If S and T are systems with isomorphic system dependence graphs then S and T are strongly equivalent.*

This theorem is an extension of the (single procedure) Equivalence Theorem [7], which states that two programs (without procedures or procedure calls) that have isomorphic *program* dependence graphs are strongly equivalent.

To prove the Multi-Procedure Equivalence Theorem we reduce the question of system-dependence-graph equivalence to that of program-dependence-graph equivalence. For terminating executions of a system this is done by rewriting the system into an equivalent program without call statements. The correctness of the rewriting is demonstrated by the Expansion Lemma, the Aliasing Information Lemma, the Path Blocking Lemma, and the Flattening Lemma.

The Expansion Lemma demonstrates two properties: (1) given two systems with isomorphic system dependence graphs, the “in-line” expansions of corresponding procedure call statements in the main programs of the two systems produce systems with isomorphic system dependence graphs; and (2) if $S'$ is the system resulting from the expansion of a call site in system $S$ then $S$ and $S'$ are strongly equivalent.

The Aliasing Information Lemma is used to prove part (1) of the Expansion Lemma. The Aliasing Information Lemma relates the procedure dependence graphs of two procedures under different sets of aliases. In particular, if two procedures have isomorphic procedure dependence graphs given some set of aliases, then the two procedures will continue to have isomorphic procedure dependence graphs given a subset of the set of aliases. The Path Blocking Lemma is also used to prove part (1) of the Expansion Lemma. The Path Blocking Lemma relates paths in the control-flow graph for the main program of system $S$ with paths in the control-flow graph for the main program of system $T$. If one of a particular set of definitions occurs on every path from $a$ to $b$ in the control-flow graph for the main program of $S$, the Path Blocking Lemma asserts that one of the corresponding set of definitions occurs on every path from the

[7] Recall that the final state is defined only on variables that occur in the *end* statement of the main program.

node corresponding to $a$ to the node corresponding to $b$ in the control-flow graph for the main program of $T$.

The Flattening Lemma demonstrates that for an arbitrary fixed initial state $\sigma$ on which system $S$ terminates, a system $\bar{S}$ can be constructed by expanding call sites in $S$ such that $S$ and $\bar{S}$ are strongly equivalent, and $\bar{S}$ makes no procedure calls when initiated on $\sigma$. The lemma also demonstrates that if systems $S$ and $T$ have isomorphic system dependence graphs and $\bar{T}$ is constructed by a parallel expansion of $T$, then $\bar{S}$ and $\bar{T}$ have isomorphic system dependence graphs.

To prove the Multi-Procedure Equivalence Theorem we argue by cases depending on whether $S$ terminates when initiated on an arbitrary state.

(1) If $S$ terminates, the Flattening Lemma is used to prove the existence of two systems strongly equivalent to $S$ and $T$ that are in the language used by Horwitz et al (*i.e.* that contain only main programs and no procedure calls) and that have isomorphic program dependence graphs. This reduces the question of system-dependence-graph equivalence to that of program-dependence-graph equivalence.

(2) If $S$ diverges, we show that $T$ also diverges.

### 3.1. The Aliasing Information Lemma

Reference parameters introduce the possibility of aliasing into a system. Variables $x$ and $y$ of procedure $P$ are potential aliases if they both refer to the same memory location during some execution of procedure $P$. The *alias information* associated with procedure $P$ is the set of pairs $(x, y)$ such that $x$ and $y$ are potential aliases in $P$. Note that the presence of the pair $(x, y)$ in the alias information for procedure $P$ implies only that there potentially exists an execution of procedure $P$ in which $x$ and $y$ both refer to the same memory location. It does not imply that $x$ and $y$ must refer to the same memory location in all executions of procedure $P$ [1,2].

Our first lemma concerns the procedure dependence graphs that would be constructed for two procedures given different sets of aliasing information. Consider two procedures $P$ and $Q$ that have isomorphic PDGs given some set of alias information $A$. The lemma asserts that if aliases are removed from $A$, procedures $P$ and $Q$ will continue to have isomorphic procedure dependence graphs. A corollary of the Aliasing Information Lemma states that adding aliases to the alias information for a procedure does not remove edges from the procedure's PDG.

*LEMMA*. (ALIASING INFORMATION LEMMA). *If procedures P and Q have isomorphic procedure dependence graphs under alias information A, and A′ is a subset of A, then procedures P and Q have isomorphic procedure dependence graphs under alias information A′.*

*PROOF*. Let $\psi$ represent the isomorphism between $PDG_P$ and $PDG_Q$ under alias information $A$. Changes in alias information do not affect the vertex sets of $PDG_P$ and $PDG_Q$. We will show that for every edge $a \longrightarrow b$ in $PDG_P$ under $A'$ there is an edge $\psi(a) \longrightarrow \psi(b)$ in $PDG_Q$ under $A'$ (by symmetry, for every edge $\psi(a) \longrightarrow \psi(b)$ in $PDG_Q$, there is an edge $a \longrightarrow b$ in $PDG_P$).

Control Edges.

Aliasing does not affect the nesting structure of a program, therefore $PDG_P$ and $PDG_Q$ have isomorphic control dependence subgraphs under $A'$.

Flow Dependence Edges.

We must show that, for every flow-dependence edge $a \longrightarrow_f b$ in $PDG_P$ under $A'$, there is an edge $\psi(a) \longrightarrow_f \psi(b)$ in $PDG_Q$ under $A'$. First we show that the edge $a \longrightarrow_f b$ is also in $PDG_P$ under $A$. In this case (because $PDG_P \approx PDG_Q$ under $A$), the edge $\psi(a) \longrightarrow_f \psi(b)$ is in $PDG_Q$ under $A$; in a second step we show that this implies that the edge $\psi(a) \longrightarrow_f \psi(b)$ is also in $PDG_Q$ under $A'$. Note that, by the definition of flow-dependence edges, vertex $a$ represents the definition of a variable $x$, vertex $b$ represents the use of a variable $y$, and $(x, y)$ is in $A'$.

1. The presence of the edge $a \longrightarrow_f b$ in $PDG_P$ under $A'$ means that there is a path in $P$'s control-flow graph from vertex $a$ to vertex $b$, along which neither $x$ nor $y$ is defined. Adding aliases to $A'$ (changing from $A'$ to $A$) does not change the existence of this path; since $(x, y)$ is in $A$, there must be a flow-dependence edge $a \longrightarrow_f b$ in $PDG_P$ under $A$.

2. Because $PDG_P \approx PDG_Q$ under $A$, the flow-dependence edge $\psi(a) \longrightarrow_f \psi(b)$ is in $PDG_Q$ under $A$. The presence of this edge means that there is a path in $Q$'s control-flow graph from $\psi(a)$ to $\psi(b)$ along which neither $x$ nor $y$ is defined. Removing aliases from $A$ (changing from $A$ to $A'$) does not change the existence of this path; since $(x, y)$ is in $A'$, there must be an edge $\psi(a) \longrightarrow_f \psi(b)$ in $PDG_Q$ under $A'$.

Def-Order Edges.

We must show that, for every def-order edge $a \longrightarrow_{do(c)} b$ in $PDG_P$ under $A'$, there is an edge $\psi(a) \longrightarrow_{do(\psi(c))} \psi(b)$ in $PDG_Q$ under $A'$. The proof is similar to the one for flow-dependence edges. Again, by the definition of def-order edges, vertex $a$ represents the definition of a variable $x$, vertex $b$ represents the definition of a variable $y$, vertex $c$ represents the use of a variable $z$, and the alias pairs $(x, y)$, $(x, z)$, and $(y, z)$ must all be in $A'$.

1. The presence of the def-order edge $a \longrightarrow_{do(c)} b$ in $PDG_P$ under $A'$ means that there are flow-dependence edges $a \longrightarrow_f c$ and $b \longrightarrow_f c$ in $PDG_P$ under $A'$. By the argument given above, these flow-dependence edges are also in $PDG_P$ under $A$. All alias pairs in $A'$ are also in $A$, and changes in alias information do not affect the relative positions of vertices $a$ and $b$; thus, the def-order edge $a \longrightarrow_{do(c)} b$ must be in $PDG_P$ under $A$.

2. Because $PDG_P \approx PDG_Q$ under $A$, the def-order edge $\psi(a) \longrightarrow_{do(\psi(c))} \psi(b)$ is in $PDG_Q$ under $A$. By the flow-dependence-edge argument the edges $\psi(a) \longrightarrow_f \psi(c)$ and $\psi(b) \longrightarrow_f \psi(c)$ are in $PDG_Q$ under $A'$. The change in alias information from $A$ to $A'$ does not affect the relative positions of $\psi(a)$ and $\psi(b)$; thus, the def-order edge $\psi(a) \longrightarrow_{do(\psi(c))} \psi(b)$ must be in $PDG_Q$ under $A'$.

Thus each edge in $PDG_P$ under alias information $A'$ has a corresponding edge in $PDG_Q$ under $A'$. By symmetry, procedures $P$ and $Q$ have isomorphic procedure dependence graphs under $A'$. □

COROLLARY. If $a \longrightarrow b$ is in $PDG_P$ under alias information $A'$, and $A'$ is a subset of $A$, then $a \longrightarrow b$ is in $PDG_P$ under $A$.

*PROOF*. The corollary follows from the control edge argument and the first cases of the flow and def-order edge arguments in the proof of the preceding lemma. □

### 3.2. The Path Blocking Lemma

The Path Blocking Lemma relates control-flow graph paths for procedures with isomorphic PDGs. In particular, if $P$ and $Q$ have isomorphic PDGs, the lemma provides the conditions under which, if all paths from node $a$ to node $b$ in $P$'s control-flow graph are "blocked" by a set of definitions $D$, then all paths from the node that corresponds to $a$ to the node that corresponds to $b$ in $Q$'s control-flow graph are blocked by the corresponding set of definitions. (A set of definitions *blocks* all control-flow graph paths from $a$ to $b$ if at least one of the definitions occurs on every path from $a$ to $b$.)

The Path Blocking Lemma relies on the Statement Blocking Lemma, which relates control-flow graph paths through individual statements or regions. The reference to "corresponding statements" of procedures $P$ and $Q$ (with isomorphic PDGs) in the Statement Blocking Lemma refers to the correspondence defined by the isomorphism for individual statements, with the obvious extension for regions.

*LEMMA.* (STATEMENT BLOCKING LEMMA). *Assume procedures $P$ and $Q$ have isomorphic procedure dependence graphs. Let $S$ be a statement or region in $P$, and $S'$ be the corresponding statement or region in $Q$. If a set of definitions blocks all control-flow graph paths from the beginning of $S$ to the end of $S$, then the corresponding set of definitions in $Q$ blocks all paths from the beginning of $S'$ to the end of $S'$.*

*PROOF.* The proof is by structural induction. First, we note that $S$ can be neither a while-loop nor a list of while-loops: Because a while-loop may execute zero times, there is a path from the beginning of a while-loop to its end that includes only the loop predicate. It is not possible for a definition to block this path; thus $S$ cannot be a while-loop. (Similarly, there is a path from the beginning of a list of while-loops to the end of the list that includes only loop predicates; thus $S$ cannot be a list of while-loops.) Now we present cases for the other four statement types.

*Case 1.* $S$ is an assignment statement. The path from the beginning of $S$ to the end of $S$ contains a single node, therefore a single definition blocks all paths through $S$. The isomorphism between $PDG_P$ and $PDG_Q$ implies that the corresponding definition blocks all paths through $S'$.

*Case 2.* $S$ is a conditional statement. For the set of definitions to block all paths through $S$, they must block all paths through the **true** branch of $S$ and all paths through the **false** branch of $S$. By the inductive hypothesis all paths through the **true** branch of $S'$ and all paths through the **false** branch of $S'$ are blocked by the corresponding set of definitions. Because all paths through $S'$ must contain either the **true** branch or the **false** branch of $S'$, the set of definitions in $Q$ blocks all paths through $S'$.

*Case 3.* $S$ is a call statement. There is a single path through the control-flow graph representation of $S$. Therefore a single definition is sufficient to block all paths through statement $S$. The isomorphism between $PDG_P$ and $PDG_Q$ implies that the corresponding definition in $Q$ blocks all paths through $S'$.

*Case 4.* $S$ is a region (statement list). For all paths through a list of statements to be blocked by a set of definitions it is necessary and sufficient that the set of definitions block all paths through one of the non-while-loop statements in the statement list. Therefore, the inductive hypothesis implies that the set of corresponding definitions in $Q$ block all paths through $S'$.

We conclude that all paths from the beginning of $S'$ to the end of $S'$ are blocked. □

The Statement Blocking Lemma deals with a path from the beginning of a statement to the end of the same statement. The Path Blocking Lemma generalizes the Statement Blocking Lemma to paths between two nodes that are not necessarily at the beginning and end of some statement. In the Path Blocking Lemma, however, it is required that there exist a flow dependence edge from the vertex at the beginning of the path to the vertex at the end of the path.

*LEMMA.* (PATH BLOCKING LEMMA). *Assume procedures $P$ and $Q$ have isomorphic procedure dependence graphs; let $\psi$ represent this isomorphism. If procedure $P$ contains a definition $a$, of a variable $x$, and a use $b$ such that (1) $a \longrightarrow_f b$, and (2) every control-flow graph path from $a$ to $b$ includes a definition $d_i$ that defines an alias of $x$, such that $d_i \longrightarrow_f b$, then in $Q$'s control-flow graph, every path from $\psi(a)$ to $\psi(b)$ contains a definition $\psi(d_i)$.*

*PROOF* (Sketch). Let $D$ be the set of definitions $d_i$ in procedure $P$. For $D$ to block all paths from $a$ to $b$, it is necessary that, on every path from $a$ to $b$, there is some statement $S_i$ such that $D$ blocks all paths from the beginning of $S_i$ to the end of $S_i$. It can be shown (using cases on the relationship between $a$ and $b$ in the control-dependence subgraph of procedure $P$'s PDG) that in fact if $D$ blocks all paths from $a$ to $b$, there is a single statement $S$ that appears on all paths from $a$ to $b$ such that $D$ blocks all paths from the beginning of $S$ to the end of $S$ (this part of the proof is omitted for brevity).

The Statement Blocking Lemma ensures that the set of definitions $\psi(D)$ in $Q$ blocks all paths through $S'$ (the statement corresponding to $S$). It can be shown (again using the control dependence subgraph of procedure $P$'s PDG and the fact that the flow edges $\psi(a) \longrightarrow_f \psi(b)$ and, $\forall d \in D, \psi(d) \longrightarrow_f \psi(b)$, imply a def-order edge $\psi(a) \longrightarrow_{do(\psi(b))} \psi(d_i)$ or $\psi(d_i) \longrightarrow_{do(\psi(b))} \psi(a)$ for some $i$) that $S'$ occurs on all paths from $\psi(a)$ to $\psi(b)$. Thus, $\psi(D)$ blocks all paths from $\psi(a)$ to $\psi(b)$. □

### 3.3. Expansion

The *expansion* of the statement "**call** $P$(...)" in the main program replaces the call statement with a list of statements constructed from the body of procedure $P$. To avoid naming conflicts, variables in procedure $P$ whose names conflict with variables in the main program are renamed so that the same name is not used by both the caller and the callee. Renaming is also used to account for any aliasing introduced by the call statement: If the statement "**call** $P$(...)" aliases two or more formal parameters, then the same (new) name is used in place of all aliased formal parameters. (Recall our assumption that systems contain no global variables.)

The statement list that replaces a call statement is composed of four sections: *Transfer-in, local initialization, body*, and *transfer-out*. *Transfer-in* statements transfer values from actual parameters to formal parameters. This is accomplished by including an assignment statement of the form "(possibly renamed) formal parameter := actual parameter" for each formal parameter in $GMOD(P) \cup GREF(P)$. *Local initialization* statements initialize local variables; there is one *local initialization* statement of the form "(possibly renamed) local variable := 0" for each local variable of $P$ that may be referenced before being defined. The *body* statements are the statements from the body of procedure $P$ (with local variables and formal parameters appropriately renamed). *Transfer-out* statements are assignment statements that transfer values back to the caller's name space. There is a *transfer-out* statement for each variable in $GMOD(P)$.

There is a subtle but important difference between the *transfer-in* and *transfer-out* statements introduced in an expansion and the actual-in and actual-out nodes in the control-flow graph used to compute data dependences. Actual-in and actual-out nodes use two different names for each formal parameter (*e.g.* for parameter $x$, $x_{in}$ and $x_{out}$) so that no dependences exist between actual-in and actual-out vertices. For a given formal parameter $x$, the name $x$ is used in both the *transfer-in* and *transfer-out* statements.

*Example.* Figure 2(a) shows a system composed of a main program and a single recursive procedure. Figure 2(b) shows the system after the call statement in the main program has been expanded. The expansion is a rewriting of the program's text. The fact that the call to procedure $P$ represents an infinite recursion does not affect the expansion. Figure 2(c) classifies each of the expanded program's statements as either *old* (those that exist in the system before the expansion), *transfer-in, local initialization, body*, or *transfer-out*.

#### 3.3.1. The Expansion Lemma

*Lemma.* (Expansion Lemma). *Let S and T be systems with isomorphic system dependence graphs; let ψ represent the isomorphism. If S′ is the system constructed by expanding one call site in the main program of S, and T′ is the system constructed by expanding the corresponding call site in the main program of T, then: (1) S and S′ are strongly equivalent; (2) T and T′ are strongly equivalent; and (3) there exists an*

```
program
   a := 1
   call P (a, b)
end

procedure P (x, y)
   local a

   x := x+y
   call P (a, x)
return
```

(a)

| program | Statement Classification |
|---|---|
| $a := 1$ | *old* |
| $x := a$ | *transfer in* |
| $y := b$ | *transfer in* |
| $a' := 0$ | *local initialization* |
| $x := x+y$ | *body* |
| **call** $P(a', x)$ | *body* |
| $a := x$ | *transfer out* |
| **end** | |
| **procedure** $P(x, y)$ | |
| **local** $a$ | |
| $x := x+y$ | *old* |
| **call** $P(a, x)$ | *old* |
| **return** | |

(b) (c)

**Figure 2.** The figure shows a simple system (a), and the system after expansion of the call site in the main program (b). In (c) each of the new system's statements is classified as one of *old*, *transfer in*, *local initialization*, *body*, or *transfer out*.

*isomorphism between* $G_{S'}$ *and* $G_{T'}$.

*PROOF*. (1),(2). (Sketch). The standard execution of a procedure call using call-by-reference parameter-passing copies into a new activation record the address of each actual parameter. The called procedure then computes its results using the values associated with these addresses. The set $GMOD(P) \cup GREF(P)$ represents a safe approximation to the variables whose values are accessed by the called procedure through the addresses placed in the activation record. The set $GMOD(P)$ represents a safe approximation to the variables defined by the called procedure through addresses placed in the activation record. Therefore, all that the statements of *transfer-in* and *transfer-out* do is make explicit the variables used and the results computed by the called procedure.

If the call statement introduces an alias between two formal parameters $x$ and $y$ then the address copied into the activation record for $x$ is the same as the address copied into the activation record for $y$. This mapping of two formal parameters to the same location is preserved in the expansion by renaming $x$ and $y$ to the same new identifier, $xy$. The renaming also preserves the aliases both introduced and propagated by the call statements occurring within the expanded procedure.

We conclude that systems $S'$ and $T'$ are strongly equivalent to systems $S$ and $T$, respectively.

(3). For every procedure $P$ in system $S$, the Aliasing Information Lemma ensures that $P$ has isomorphic PDGs in $G_{S'}$ and $G_{T'}$. We must demonstrate that the PDGs of the respective main programs of $S'$ and $T'$ are isomorphic and that $G_{S'}$ and $G_{T'}$ have isomorphic call, parameter-in and parameter-out edges. We do this by first demonstrating that each vertex in $PDG_{Main}$ of $G_{S'}$ has a corresponding vertex in $PDG_{Main}$ of

$G_{T'}$. Second, we show that each call, parameter-in, and parameter-out edge in $G_{S'}$ has a corresponding edge in $G_{T'}$ and that each edge in $PDG_{Main}$ of $G_{S'}$ has a corresponding edge in $PDG_{Main}$ of $G_{T'}$. By symmetry, each vertex in $PDG_{Main}$ of $G_{T'}$ has a corresponding vertex in $PDG_{Main}$ of $G_{S'}$ and each edge of $G_{T'}$ has a corresponding edge in $G_{S'}$; thus, there is an isomorphism between $G_{S'}$ and $G_{T'}$.

*Vertices*

Assume that the call statement that was expanded in $S$ is "**call** $P$(...)". We first demonstrate the existence of a 1-1 mapping, $\psi'_0$, from the vertex set of $G_{S'}$, denoted by $V(G_{S'})$, into the vertex set of $G_{T'}$. We consider four possibilities for a vertex $v \in V(G_{S'})$: 1) $v$ is in $G_S$, 2) $v$ represents a *body* statement, 3) $v$ represents either a *transfer-in* or *transfer-out* statement, or 4) $v$ represents a *local initialization* statement.

Case 1) $v \in V(G_S)$

If $v \in V(G_S)$ then $\psi(v) \in V(G_T)$. Since the only vertices removed by the expansion are those that represent the expanded call statement and $\psi(v)$ is not any of these, a corresponding vertex exists in $G_{T'}$.

Case 2) $v$ represents a *body* statement

Vertex $v$ represents a statement (possibly with renamed variables) from procedure $P$'s body. Thus $v$ is a copy of a vertex in $PDG_P$ of $G_S$. Since the expansion of the corresponding call statement in $T$ also includes procedure $P$'s body and the same renaming is done, there is a copy of $\psi(v)$ in $G_{T'}$.

Case 3) $v$ represents either a *transfer-in* or *transfer-out* statement

By definition there is a 1-1 correspondence between the vertices that represent *transfer-in* (respectively, *transfer-out*) statements in $G_{S'}$ and the actual-in (actual-out) vertices of the expanded call site in $G_S$. The same is true in $G_{T'}$ and $G_T$. By assumption there is a 1-1 mapping from the vertices of $G_S$ onto the vertices of $G_T$. Thus the composition of these three mappings yields a 1-1 mapping from the vertices representing *transfer-in* (*transfer-out*) statements of $S'$ into the vertices representing *transfer-in* (*transfer-out*) statements of $T'$.

Case 4) $v$ represents a *local initialization* statement

In both $S$ and $T$ the same set of local variables may be referenced before being defined in procedure $P$. Therefore there is a 1-1 correspondence between the *local initialization* vertices in $G_{S'}$ and $G_{T'}$.

*Edges*

We demonstrate that the edge sets are isomorphic by considering the four edge types separately.

Case (I). Interprocedural Dependence Edges.

Let $a \longrightarrow_i b$ be an interprocedural dependence edge in $G_{S'}$. By definition, interprocedural dependence edges connect corresponding pairs of vertices (actual-in vertices to formal-in vertices, formal-out vertices to actual-out vertices, and call site vertices to procedure entry vertices). Therefore, there is an interprocedural dependence edge $\psi'_0(a) \longrightarrow_i \psi'_0(b)$ in $G_{T'}$ corresponding to $a \longrightarrow_i b$.

Let $\psi'_1$ represent the isomorphism between the interprocedural dependence edges of $G_{S'}$ and $G_{T'}$.

Case (II). Control Dependence Edges.

Let $e' \in E(G_{S'})$ be a control dependence edge. ($E(G)$ denotes the edge set of graph $G$.) We consider two possibilities for $e'$.

Case 1) $e' \in E(G_S)$

The isomorphism between $G_S$ and $G_T$ implies $\psi(e') \in E(G_T)$. The expansion does not change the nesting structure of statements; thus $\psi(e') \in E(G_{T'})$.

Case 2) $e' \notin E(G_S)$

The source of edge $e'$ is either the vertex $w$ on which the expanded call statement was control dependent, or a vertex that is a copy of a vertex in $PDG_P$. In both cases, $e'$ corresponds to an edge $e$ in $PDG_P$. This correspondence is as follows: If the source of $e'$ is $w$ then the source of $e$ is procedure $P$'s entry vertex. If the target of $e'$ is a vertex representing a *transfer-in* or *transfer-out* statement then the target of $e$ is a formal-in or formal-out vertex, respectively. If the target of $e'$ is a vertex that represents a local initialization statement then the target of $e$ is an initial definition vertex. Otherwise the end-points of $e'$ are copies of vertices from procedure $P$'s PDG. Edge $e \in E(G_S)$ therefore $\psi(e) \in E(G_T)$. In $G_{T'}$ the expansion introduces a copy of $\psi(e)$ corresponding to $e'$.

Let $\psi'_2$ represent the isomorphism between the control dependence edges of $G_{S'}$ and $G_{T'}$.

Case (III). Flow Dependence Edges.

To show that the PDGs for the main programs of $S'$ and $T'$ have isomorphic flow dependence edges, we classify flow dependence edges based on their end-points (Table 1 shows which end-point combinations are possible). An edge is in one of two classes:

| | $a \longrightarrow_f b$ | *b*: *old* | *transfer in* | *local initialization* | *body* | *transfer out* |
|---|---|---|---|---|---|---|
| *a* | *old* | 1 | 1 | - | - | - |
| | *transfer in* | - | - | - | 2 | 2 |
| | *local initialization* | - | - | - | 2 | - |
| | *body* | - | - | - | 2 | 2 |
| | *transfer out* | 1 | 1 | - | - | - |

"1" — Class 1 edges
"2" — Class 2 edges
"-" — either by construction or by the use of renaming, an edge with these end points is not possible.

**Table 1.** Table 1 shows which end-point combinations are possible and which are not possible. For example, *old* $\longrightarrow_f$ *local initialization* is not possible because, by construction, *local initialization* statements reference no variables; *transfer in* $\longrightarrow_f$ *transfer in* is not possible because *transfer in* statements assign to variables in one name space and reference variables from the other.

Class 1: *old* $\longrightarrow$ *old*, *old* $\longrightarrow$ *transfer-in*, *transfer-out* $\longrightarrow$ *old*, *transfer-out* $\longrightarrow$ *transfer-in*

Class 2: *transfer-in* $\longrightarrow$ *body*, *transfer-in* $\longrightarrow$ *transfer-out*, *body* $\longrightarrow$ *body*, *body* $\longrightarrow$ *transfer-out*,
*local initialization* $\longrightarrow$ *body*

We consider Class 1 and Class 2 edges separately. In each case assume $e' = a' \longrightarrow_f b' \in E(G_{S'})$ where $a'$ defines a variable $x$ that is used at $b'$ (there is no aliasing in the main program). Let $CFG_S$ denote the control-flow graph for the main program of system $S$ and let $s$ denote the call statement in $S$ that has been expanded in $S'$.

Case 1) $e'$ is in Class 1

For class 1 edges we first demonstrate that edge $e'$ corresponds to an edge $e = a \longrightarrow_f b \in E(G_S)$. We do this by observing that the end-points of $e'$ correspond to vertices

$a$ and $b$ in $G_S$, and then demonstrating that an $x$-definition-free path exists from $a$ to $b$ in $CFG_S$. By assumption, edge $e$ has a corresponding edge $\psi(e)$ in $G_T$. The final step is to demonstrate that there is an edge in $G_{T'}$ corresponding to $\psi(e)$.

We demonstrate that $e'$ corresponds to $e$ by showing that $a$ assigns to $x$, $b$ uses $x$, and there exists an $x$-definition-free path from $a$ to $b$. First consider the end-points of $e'$. If $a'$ represents a *transfer-out* statement of $S'$, then $a$ is an actual-out vertex in $G_S$. $a$ assigns to $x$, because a *transfer-out* statement assigns to the same variable as the corresponding actual-out vertex. If $b'$ represents a *transfer-in* statement of $S'$, then $b$ is an actual-in vertex in $G_S$. $b$ references $x$, because a *transfer-in* statement references the same variables as the corresponding actual-in vertex. The remaining possibility is that $a$ or $b$ represent *old* statements but *old* statement are unchanged by the expansion. We conclude that $a$ represents a definition of $x$ and that $b$ represents a use of $x$. We must now demonstrate that there is an $x$-definition-free path in $CFG_S$ from $a$ to $b$.

Edge $e'$ implies the existence of an $x$-definition-free path $p'$ from $a'$ to $b'$ in $CFG_{S'}$. If $p'$ does not contain any of the nodes representing the statements added by the expansion then $p'$ occurs in $CFG_S$. Otherwise, the nodes added by the expansion can appear on $p'$ in one of two places: (1) in the middle of $p'$ but not at either end of $p'$, or (2) at either one or both ends of $p'$. For (1) $p'$ is composed of five segments that contain nodes representing: (a) *old* statements, (b) *transfer-in* statements, (c) *local initialization* and *body* statements, (d) *transfer-out* statements, and (e) *old* statements. For (2) $p'$ is composed of three (possibly empty) segments that contain nodes representing: (f) *transfer-out* statements, (g) *old* statements, and (h) *transfer-in* statements. These segments have corresponding segments in $CFG_S$, thus there is a path in $CFG_S$ corresponding to $p'$. The correspondence is as follows: (a), (e), and (g) exist unchanged in $CFG_S$, (b) and (h) correspond to segments composed of actual-in nodes in $CFG_S$, (d) and (f) correspond to segments composed of actual-out nodes in $CFG_S$, and (c) corresponds to a null (empty) segment in $CFG_S$. We now demonstrate that the path in $CFG_S$ is also $x$ definition free. Segments (a), (e), and (g) are unchanged in $CFG_{S'}$, therefore they are $x$-definition-free. In $CFG_S$ (b) and (h) are composed of actual-in nodes. Because actual-in nodes assign to intermediate temporary variables their existence on the path will not kill a definition of $x$. Finally, in $CFG_S$ (d) and (f) are composed of actual-out nodes. Because actual-out nodes represent assignments to the same variables as *transfer-out* statements and $p'$ does not kill $x$, the actual-out nodes on the path in $CFG_S$ do not kill $x$. The existence of an $x$-definition-free path from $a$ to $b$ in $CFG_S$ implies there is an edge $e = a \longrightarrow_f b$ in $E(G_S)$.

$G_S$ is isomorphic to $G_T$, therefore there is an edge $\psi(e) \in E(G_T)$. This edge implies there is an $x$-definition-free path $p$ from $\psi(a)$ to $\psi(b)$ in $CFG_T$. We demonstrate that an $x$-definition-free path exists from $\psi'_0(a)$ to $\psi'_0(b)$ in $CFG_{T'}$. If the path in $CFG_T$ does not contain any of the nodes that represent the call statement corresponding to $s$ then the same path occurs in $CFG_{T'}$. Otherwise, the nodes added by the expansion can appear on $p$ in one of two places: (1) in the middle of $p$ but not at either end of $p$, or (2) at either one or both ends of $p$. For (1) $p$ is composed of four segments that contain the following: (a) nodes representing *old* statements, (b) actual-in nodes, (c) actual-out nodes, and (d) nodes representing *old* statements. For (2) $p$ is composed of three (possibly empty) segments that contain the following: (e) actual-out nodes, (f) nodes representing *old* statements, and (g) actual-in nodes. These segments have corresponding segments in $CFG_{T'}$, thus there is a path in $CFG_{T'}$ corresponding to $p$ (for a path containing (b) and (c) the corresponding path in $CFG_{T'}$ also includes nodes representing *local-initialization*, and *body* statements). The correspondence is as follows: (a), (d), and (f) exist unchanged in $CFG_{T'}$, (b) and (g) correspond to segments composed of nodes representing *transfer-in* statements in $CFG_{T'}$, (c) and (e) correspond to segments composed of nodes representing *transfer-out* statements in $CFG_{T'}$. We now demonstrate that the path in $CFG_{T'}$ is also *x* definition free. Segments (a), (d), and (f) are unchanged in $CFG_{T'}$, therefore they are *x*-definition-free. In $CFG_{T'}$ (b) and (g) are composed of nodes representing *transfer-in* statements. *Transfer-in* as well as *local-initialization*, and *body* statements assign to variables in procedure *P*'s name space. If *P* has a parameter or local variable named *x*, then that variable is renamed as part of the expansion, therefore nodes representing *transfer-in*, *local-initialization*, and *body* statements do not kill *x*. Finally, in $CFG_{T'}$ (c) and (e) are composed of nodes representing *transfer-out* nodes. Because *transfer-out* statements assign to the same variables as actual-out nodes and *p* does not kill *x*, the nodes representing *transfer-out* statements on the path in $CFG_{T'}$ do not kill *x*. The *x*-definition-free path from $\psi'_0(a)$ to $\psi'_0(b)$ implies there is an edge $\psi'_0(a) \longrightarrow_f \psi'_0(b) \in E(G_{T'})$ that corresponds to $e'$.

Case 2) $e'$ is in Class 2

As in the proof of Case 1, we first show that $e'$ corresponds to an edge $e = a \longrightarrow_f b \in E(G_S)$. By assumption, $\psi(e) \in E(G_T)$. Arguing by contradiction, we show that the expansion introduces a corresponding edge in $G_{T'}$. Although the proof for Case 2 is similar to the proof of Case 1, the edge *e* in Case 1 occurs in the main program's PDG, while in Case 2, *e* occurs in the PDG for procedure *P*. This make the proof more difficult because of the possibility of aliasing in procedure *P*.

In general, the aliasing information for procedure *P* is a superset of that introduced at *s*. However, the Corollary to the Aliasing Information Lemma tells us that if *e* exists under the assumption that the only aliases are those introduced by call statement *s*, then *e* will continue to exist in the presence of additional aliases. Therefore, we need only demonstrate that $e'$ corresponds to an edge *e* in $PDG_P$ assuming that the only aliases are those introduced by *s*.

We demonstrate that $e'$ corresponds to *e* by showing that *a* assigns to a variable *x*, *b* references *y* (an alias of *x*), and there exists a path from *a* to *b* on which neither *x* nor *y* are redefined. The proof of Case 2 is complicated by the renaming that occurs during expansion to account for aliasing introduced at *s*. We first consider the end-points of $e'$ assuming that no renaming because of aliasing has occurred at $a'$ or $b'$ (thus *x* and *y* are the same variable). We then factor in the effects of the renaming.

If $a'$ represents a *local initialization* statement then *a* is the corresponding initial-definition vertex that assigns to *x*. If $a'$ represents a *transfer-in* statement of $S'$, then *a* is a formal-in vertex in $G_S$. *a* assigns to *x*, because a *transfer-in* statement assigns to the same variable as the corresponding formal-in vertex. If $b'$ represents a *transfer-out* statement of $S'$, then *b* is a formal-out vertex in $G_S$. *b* references *x*, because a *transfer-out* statement references the same variable as the corresponding formal-out vertex. The remaining possibility is that *a* or *b* are *body* statements but *body* statements are copies of the statements in procedure *P* and therefore are unchanged by the expansion.

We now consider the effects of renaming on *x* and *y*. If $a'$ assigns to a renamed variable (*e.g. xy*) then $b'$ references that same renamed variable. In system *S*, *a* assigns to one of the aliased components (*e.g. x*) of the variable assigned to at $a'$ and *b* references one of the aliased components (*e.g. y*). In addition, call statement *s* must introduce an alias between these variables (*i.e. x* and *y*).

Thus, $a$ represents a definition of $x$, $b$ represents a reference of $y$, and, either $s$ introduces an alias between $x$ and $y$ or $x$ and $y$ are the same variable. To show that a definition-free path exists from $a$ to $b$ we first observe that a copy of the control-flow graph for procedure $P$ appears unchanged (except for node renaming) in $CFG_{S'}$, therefore there is a control-flow graph path from $a$ to $b$ in the control-flow graph for procedure $P$ in system $S$. This path is free of any assignments to either $x$ or $y$: An assignment to $x$ (or $y$) on this path would be subject to the renaming. The renamed assignment would kill the assignment at $a'$, thus no such assignment can exist. Therefore there is an edge $e = a \longrightarrow_f b$ in $PDG_P$ of $G_S$.

$G_S$ is isomorphic to $G_T$, therefore $\psi(e)$ is an edge in $PDG_P$ of $G_T$. This edge is shown in Figure 4(a). We demonstrate that a corresponding edge exists in $G_{T'}$ by contradiction; assume there is no corresponding edge in $G_{T'}$. If no edge $\psi'_0(a') \longrightarrow_f \psi'_0(b')$ exists in $G_{T'}$ then the definition at $\psi'_0(a')$ must be killed on all control-flow graph paths from $\psi'_0(a')$ to $\psi'_0(b')$ by an assignment at some node corresponding to a vertex $\psi_0{}'(c_i{}')$ (where $\psi_0{}'(c_i{}')$ assigns to $z_i$ and the call-site corresponding to $s$ aliases $z_i$ and $x$.) Assuming there are three paths from $\psi(a)$ to $\psi(b)$ the relevant portion of $T'$'s system dependence graph is shown in Figure 4(b). The $\psi(c_i)$s block all the paths from $\psi(a)$ to $\psi(b)$ in $CFG_T$, therefore because $\psi(a) \longrightarrow_f \psi(b) \in E(G_T)$, the Path Blocking Lemma a $c_i$ exists on every control-flow graph path from $a$ to $b$ in $CFG_S$. In systems $S$ and $T$ call statement $s$ introduces the same aliases. Because $s$ aliases $z_i$ and $x$ in $S$, in $CFG_{S'}$, each $c_i$ represents a kill of the definition at $a'$. A $c_i{}'$ exists on all paths from $a'$ to $b'$, thus the definition at $a'$ is killed by the definition at some $c_i{}'$ on all paths from $a'$ to $b'$ in $CFG_{S'}$; this is a contradiction because we have assumed there is an edge $a' \longrightarrow_f b'$ in $G_{S'}$. Therefore there must exist an edge in $G_{T'}$ corresponding to $e'$. (The relevant portion of $S'$'s system dependence graph is shown in Figure 4(c).)

In both cases if $e'$ is loop-carried then the backedge in $CFG_{S'}$ is also a backedge in $CFG_{T'}$, thus the edge corresponding to $e'$ is carried by the same loop.

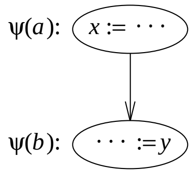
Let $\psi'_3$ represent the isomorphism between the flow dependence edges of $G_{S'}$ and $G_{T'}$.

Case (IV). Def-order Edges.

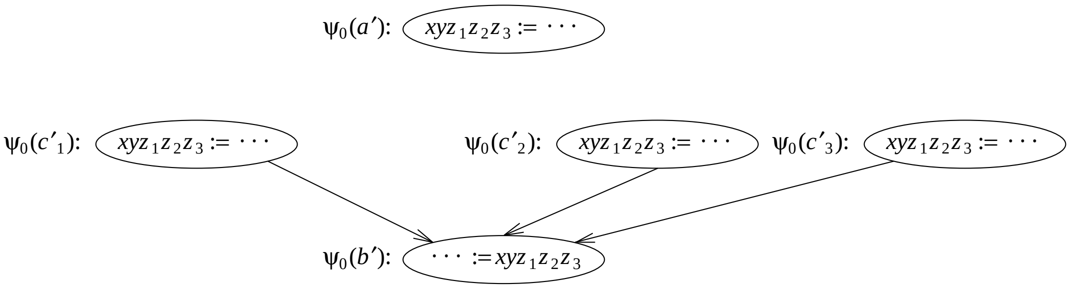
To show that the PDGs for the main programs of $S'$ and $T'$ have isomorphic def-order dependence edges, we consider two cases (recall that a def-order edge is defined in terms of two flow dependence edges with the same target): 1) the two flow dependence edges are in Class 1; 2) the two flow dependence edges are in Class 2. It is not possible that one flow dependence edge is in Class 1 and the other is in Class 2, because the target of a Class 1 edge is either an *old* or a *transfer-in* vertex, while the target of a Class 2 edge is either a *body* or *transfer-out* vertex. Let $e'$ be the def-order edge $a' \longrightarrow_{do(c')} b'$. This assumes $a'$ occurs to the left of $b'$ in the main program's abstract syntax tree, $a'$ and $b'$ are nested in the same branch of any conditional statement that encloses both of them, and $a' \longrightarrow_f c'$ and $b' \longrightarrow_f c'$ are flow edges such that $a'$ and $b'$ define variable $x$ and $c'$ references $x$.

Case 1) $a' \longrightarrow_f c'$ and $b' \longrightarrow_f c'$ are in Class 1

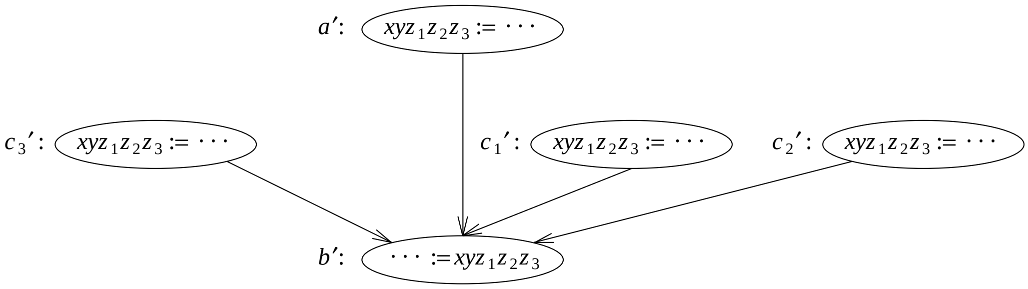
The first half of the argument for Class 1 flow dependence edges (Case III above) implies that $a \longrightarrow_f c$ and $b \longrightarrow_f c$ are flow dependence edges in $G_S$. Because the expansion does not change the relative positions (in the abstract syntax tree) of $a$ and $b$, a def-order edge $e = a \longrightarrow_{do(c)} b$ corresponding to $e'$ exists in $G_S$. $G_S$ is isomorphic to $G_T$, thus a def-order edge $\psi(e)$ exists in $G_T$. By Case III above, $\psi'_3(a' \longrightarrow_f c')$ and $\psi'_3(b' \longrightarrow_f c')$ are flow-dependence edges in $G_{T'}$. Because the expansion does not change the relative positions of $\psi(a)$ and $\psi(b)$, there is a def-order edge corresponding to $\psi(e)$ (and thus to $e'$) in $G_{T'}$.




(a). The flow edge $\psi(a) \longrightarrow_f \psi(b)$ exists in $G_T$, therefore there must be a control-flow graph path from $\psi(a)$ to $\psi(b)$ that contains no assignments to $x$ or $y$.




(b). If there is no flow edge $\psi'_0(a') \longrightarrow_f \psi'_0(b')$ in $G_{T'}$ then there must exist assignments (at $\psi'_0(c'_i)$) on all control-flow graph paths from $\psi'_0(a')$ to $\psi'_0(b')$ in $CFG_{T'}$. This example assumes three such paths exist. Variables $x$, $y$, $z_1$, $z_2$, and $z_3$ are all renamed $xyz_1z_2z_3$ because they are all aliased by statement $s$.




(c). The flow edge $a' \longrightarrow_f b'$ exists in $G_S'$. However, there are assignments to the renamed variable ($xyz_1z_2z_3$) on all paths from $a'$ to $b'$. This is a contradiction.

**Figure 4.** The contradiction in case 2 of the flow dependence argument of the expansion lemma.

Case 2) $a' \longrightarrow_f c'$ and $b' \longrightarrow_f c'$ are in Class 2

The first half of the argument for Class 2 flow dependence edges (Case III above) implies that $a \longrightarrow_f c$ and $b \longrightarrow_f c$ are flow dependence edges in $PDG_P$ of $G_S$. Because the expansion does not change the relative positions of $a$ and $b$, a def-order edge $e = a \longrightarrow_{do(c)} b$ corresponding to $e'$ exists in $PDG_P$ of $G_S$. $G_S$ is isomorphic to $G_T$, thus a def-order edge $\psi(e)$ exists in $G_T$. By Case III above, $\psi'_3(a' \longrightarrow_f c')$ and $\psi'_3(b' \longrightarrow_f c')$ are flow-dependence edges in $G_{T'}$. Because the expansion does not change the relative positions of $\psi(a)$ and $\psi(b)$, there is a def-order edge corresponding to $\psi(e)$ (and thus to $e'$) in $G_{T'}$.

Let $\psi'_4$ represent the isomorphism between the def-order edges of $G_{S'}$ and $G_{T'}$.

We have demonstrated that every edge in $G_{S'}$ corresponds to an edge in $G_{T'}$. In each case the symmetric argument shows that each edge in $G_{T'}$ corresponds to an edge in $G_{S'}$. We have also demonstrated that these edges have the correct endpoints. Therefore the two graphs $G_{S'}$ and $G_{T'}$ are isomorphic (the isomorphism is the conjunction of $\psi'_1$, $\psi'_2$, $\psi'_3$, and $\psi'_4$). □

COROLLARY. (BODYLESS EXPANSION COROLLARY). *Let S and T be two systems with isomorphic system dependence graphs. If a call statement in the main program is replaced by only transfer-in and transfer-out statements, the third assertion of the lemma still holds. That is, the two graphs $G_{S'}$ and $G_{T'}$ are isomorphic.*

*PROOF*. For procedures in $S'$ and $T'$ the result follows from the Aliasing Information Lemma. For the main program the result follows from the proof of the Expansion Lemma with a straightforward simplification for Class 2 edges.

### 3.4. The Flattening Lemma

The Flattening Lemma demonstrates that for a particular initial state, it is possible to transform a program with procedure calls into an equivalent program in which no procedure calls are executed.

*Definition*. The *invocation tree* for the execution of a program (on some initial state) represents the procedure invocations made during the execution. Nodes of the tree represent particular activations of procedures. Edges represent the procedure calls (invocations) made during each activation. The *size* of an invocation tree is the number of nodes in the tree.

*Definition*. For a particular invocation of a procedure (*i.e.* a node in the invocation tree) we classify each call site in the corresponding procedure as either *used* or *unused*. A *used* call site in activation *a* is a call site that is responsible for at least one of *a*'s children in the invocation tree. Used call sites are executed at least once during the particular invocation of the procedure. An *unused* call site in activation *a* is a call site that gives rise to none of *a*'s children in the invocation tree.

Note that it is possible for a call site in a program to be both used and unused at different nodes in the invocation tree.

*LEMMA*. (FLATTENING LEMMA). *Let S and T be systems with isomorphic system dependence graphs. If $\sigma$ is a state on which S terminates then there exist programs $\bar{S}$ and $\bar{T}$ such that: (1) S are $\bar{S}$ are strongly equivalent; (2) T and $\bar{T}$ are strongly equivalent; (3) the size of the invocation tree for $\bar{S}$ when initiated on $\sigma$ is 1; and (4) $\bar{S}$ and $\bar{T}$ have isomorphic system dependence graphs.*

*PROOF*. Let *IT* be the invocation tree for *S* when initiated on state $\sigma$. Because *S* terminates, *IT* is finite. Let *callsite* be a used call site in the root activation of *IT* (the root activation corresponds to the initial execution of the main program); suppose that *callsite* represents a call to procedure *P*. We construct from *S* a new system $S_1$ by expanding procedure *P* at *callsite*. We construct $T_1$ by expanding *T* at the corresponding call site.

The key observation is that the invocation tree for $S_1$ is smaller on $\sigma$ than *IT*. This is because all invocations of procedure *P* from *callsite* have been removed and no new procedure invocations have been introduced. *Callsite* is a used call site, therefore at least one such invocation exists. (If *callsite* is enclosed in a while-loop, it may produce more than one child in the invocation tree.)

By repeated expansion we construct a finite sequence of systems each having a smaller invocation tree than its predecessor. Let $\bar{S}$ be the final system in this sequence (*i.e.* the size of the invocation tree for $\bar{S}$ when initiated on state $\sigma$ is 1). If $S_0 = S$ and $S_n = \bar{S}$ then by the Expansion Lemma $S_i$ and $S_{i+1}$ are strongly equivalent for $0 \le i < n$. This implies that $S$ and $\bar{S}$ are strongly equivalent. By a similar argument $T$ and $\bar{T}$

are strongly equivalent. The Expansion Lemma also implies that $G_{\bar{S}} \approx G_{\bar{T}}$. This follows from the pairs of isomorphic graphs; $G_{S_0} \approx G_{T_0}$, $G_{S_1} \approx G_{T_1}$, $\cdots$ $G_{S_n} \approx G_{T_n}$, where $G_{S_0} = G_S$, $G_{T_0} = G_T$, $G_{S_n} = G_{\bar{S}}$ and $G_{T_n} = G_{\bar{T}}$. □

### 3.5. The Multi-Procedure Equivalence Theorem

Given two systems $S$ and $T$ with isomorphic system dependence graphs, we use the results from the Flattening Lemma to construct two systems that are equivalent to $S$ and $T$, respectively, but have only main programs. This construction rewrites $S$ and $T$ into the simpler language used in [7]. The Multi-Procedure Equivalence Theorem then follows from the (single procedure) Equivalence Theorem.

*THEOREM.* (MULTI-PROCEDURAL EQUIVALENCE THEOREM). *If S and T are systems with isomorphic system dependence graphs then S and T are strongly equivalent.*

*PROOF.* The proof is by contradiction. Assume that $S$ and $T$ are inequivalent, and that σ is an initial state that demonstrates the inequivalence.

Case 1) Suppose $S$ terminates when initiated on state σ.

We will demonstrate that $T$ also terminates on σ and produces the same final state as does $S$ on σ by constructing from $S$ and $T$ two programs $\hat{S}$ and $\hat{T}$ such that:

(1) $S$ and $\hat{S}$ are equivalent on σ;
(2) $T$ and $\hat{T}$ are equivalent on σ;
(3) the main programs of $\hat{S}$ and $\hat{T}$ contain no call statements;
(4) the main programs of $\hat{S}$ and $\hat{T}$ have isomorphic *program* dependence graphs.

The (single procedure) Equivalence Theorem, together with (3) and (4) above, guarantees that programs $\hat{S}$ and $\hat{T}$ are strongly equivalent; thus, they produce the same final state when initiated on σ. This, together with (1) and (2) above, guarantees that programs $S$ and $T$ also produce the same final state when initiated on σ, contradicting the assumption that σ is a state for which $S$ and $T$ are inequivalent.

First, by the Flattening Lemma there exist programs $\bar{S}$ and $\bar{T}$ such that:

(1) $S$ and $\bar{S}$ are equivalent on σ;
(2) $T$ and $\bar{T}$ are equivalent on σ;
(3) $G_{\bar{S}} \approx G_{\bar{T}}$; and
(4) no call statement in $\bar{S}$ is executed when $\bar{S}$ is initiated on state σ.

To construct $\hat{S}$ and $\hat{T}$ from $\bar{S}$ and $\bar{T}$ we must remove the call statements from $\bar{S}$ and $\bar{T}$ while preserving the equivalence of $\bar{S}$ and $\hat{S}$, and of $\bar{T}$ and $\hat{T}$ on σ. We must also guarantee that $G_{\hat{S}} \approx G_{\hat{T}}$. We begin by observing that, because none of the call statements in $\bar{S}$ are executed when $\bar{S}$ is initiated on σ, we can replace each call statement with an arbitrary list of statements without affecting the final state. We do not know, *a priori*, that none of the call statements in $\bar{T}$ are executed when $\bar{T}$ is initiated on σ; however, by a careful choice of the list of statements with which call statements in both $\bar{S}$ and $\bar{T}$ are replaced, we can demonstrate that this is indeed the case.

We construct $\hat{S}$ and $\hat{T}$ from $\bar{S}$ and $\bar{T}$ by replacing each call statement with a list of *transfer-in* statements, an *abort* statement, and a list of *transfer-out* statements. Because an *abort* statement has no incoming or outgoing data dependence edges, the Bodyless Expansion Corollary ensures that $\hat{S}$ and $\hat{T}$ have isomorphic system dependence graphs. The main programs of $\hat{S}$ and $\hat{T}$ include no call statements, and have isomorphic *program* dependence graphs (as defined in [7]); thus, by the (single procedure) Equivalence Theorem, $\hat{S}$ and $\hat{T}$ produce the same final state when initiated on $\sigma$.

As noted above, $\bar{S}$ and $\hat{S}$ are equivalent on $\sigma$ because $\bar{S}$ executes no call statements when initiated on $\sigma$. All that remains to be shown is that $\bar{T}$ and $\hat{T}$ are equivalent on $\sigma$. This follows from the fact that $\hat{T}$ does not abort on $\sigma$ (because it is strongly equivalent to $\hat{S}$, which does not abort on $\sigma$). The only difference between $\bar{T}$ and $\hat{T}$ is the substitution of transfer-in, abort, and transfer-out statements for call statements; thus, since $\hat{T}$ executes none of its abort statements, $\bar{T}$ executes none of its call statements, and $\bar{T}$ and $\hat{T}$ are equivalent on $\sigma$.

Case 2) Suppose $S$ does not terminate when initiated on state $\sigma$.
For $\sigma$ to be a state that demonstrates the inequivalence of programs $S$ and $T$, it must be that $T$ terminates. However, then the argument given in Case 1 holds for $T$ (*i.e.* $S$ must terminate). This contradicts the assumption that $S$ does not terminate when initiated on state $\sigma$.

In either case $\sigma$ fails to demonstrate the inequivalence of $S$ and $T$. Because $\sigma$ is an arbitrary initial state we conclude that programs $S$ and $T$ are strongly equivalent. □

## 4. RELATED WORK

This paper studies the relation between program semantics and the dependence graphs used as intermediate program representations. [7] began this study with the proof of the Equivalence Theorem. In this paper we have extended this semantic foundation for dependence graphs to a language that contains procedures and procedure calls.

The data dependence edges used in this paper (as well as those in [7]) are somewhat non-standard. Ordinarily, def-order dependences are not included, but two other kinds of data dependences, called *anti-dependences* and *output dependences* are used instead.[8]

For flow dependences, anti-dependences, and output dependences, (in the absence of aliasing) a program component $v_2$ has a dependence on component $v_1$ due to variable $x$ only if execution can reach $v_2$ after $v_1$ and there is no intervening definition of $x$ along the execution path by which $v_2$ is reached from $v_1$. There is a flow dependence if $v_1$ defines $x$ and $v_2$ uses $x$; there is an anti-dependence if $v_1$ uses $x$ and $v_2$ defines $x$; there is an output dependence if $v_1$ and $v_2$ both define $x$.

Although def-order dependences resemble output dependences in that they both relate two assignments to the same variable, they are two different concepts. An output dependence, denoted $v_1 \longrightarrow_o v_2$, between two definitions of $x$ can hold only if there is no intervening definition of $x$ along some execution path from $v_1$ to $v_2$; however, there can be a def-order dependence $v_1 \longrightarrow_{do} v_2$ between two definitions even if there is an intervening definition of $x$ along *all* execution paths from $v_1$ to $v_2$. This situation is illustrated by the following example program fragment, which demonstrates that it is possible to have a program in which there is a dependence $v_1 \longrightarrow_{do} v_2$ but not $v_1 \longrightarrow_o v_2$, and *vice versa*:

[8] As with flow dependences, anti-dependence and output dependence may be further characterized as loop-independent or loop-carried.

[1] $x := 10$
[2] **if** $P$ **then**
[3] $x := 11$
[4] $x := 12$
[5] **fi**
[6] $y := x$

The one def-order dependence, [1] $\longrightarrow_{do([6])}$ [4], exists because the assignments to $x$ in lines [1] and [4] both reach the use in line [6]. In contrast, the output dependences are [1] $\longrightarrow_o$ [3] and [3] $\longrightarrow_o$ [4], but there is no output dependence [1] $\longrightarrow_o$ [4].

The Multi-Procedure Equivalence Theorem still holds if the system dependence graph is defined to have output dependence edges rather than def-order dependences edges. Because a system's def-order dependence edges can be determined given the flow dependence edges and loop-independent output dependence edges, if the system dependence graphs (having output dependence edges) of two programs are isomorphic, then their system dependence graphs (having def-order edges) are isomorphic; consequently, by the Multi-Procedure Equivalence Theorem, they are strongly equivalent.

The motivation for the (single procedure) Equivalence Theorem was as the semantic basis for the (single procedure) program integration algorithm of [6]. The Multi-Procedure Equivalence Theorem has a similar motivation; we intend to use the Multi-Procedure Equivalence Theorem as the first step in providing the semantic basis for a multi-procedure program integration algorithm.

The program integration algorithm of [6] uses program slicing to identify sub-computations of a program. The *slice* of a program is defined as a reachability problem on the program's dependence graph. The system dependence graph was introduced in [5] to compute the interprocedural slice of a program.

The definition of the system dependence graph used in this paper differs from the one given in [5] in that is does not include the *summary edges* defined in [5]. Summary edges represent a transitive path of dependence edges (both interprocedural dependence edges and internal dependence edges) from an actual-in vertex to an actual-out vertex, such that the path corresponds to a valid procedure calling sequence. (*i.e.* a path cannot contain edges that represent a procedure being called from one call site and returning to a different call site). The reason for including these edges is to simplify operations on the system dependence graphs that involve traversing paths of dependence edges. Program slicing is an example of such an operation. Summary edges do not affect the proof of the Multi-Procedure Equivalence Theorem because they can be derived from a graph without summary edges.

## REFERENCES


1. Banning, J.P., “An efficient way to find the side effects of procedure calls and the aliases of variables,” *6th annual ACM Symposium on Principles of Programming Languages*, pp. 29-41 (January 1979).

2. Cooper, K.D. and Kennedy, K., “Fast interprocedural alias analysis,” *Conference Record of the Sixteenth ACM Symposium on Principles of Programming Languages,* (Austin, TX, January 11-13, 1989), pp. 49-59 ACM, (1989).

3. Cytron, R., Hind, M., and Hsieh, W., “Automatic Generation of DAG Parallelism,” *Proceedings of the ACM SIGPLAN 89 Conference on Programming Language Design and Implementation,* (Portland, OR, June 21-23, 1889)*, ACM SIGPLAN Notices* **24**(7) pp. 54-68 (June 1989).

4. Ferrante, J., Ottenstein, K., and Warren, J., “The program dependence graph and its use in optimization,” *ACM Transactions on Programming Languages and Systems* **9**(3) pp. 319-349 (July 1987).

5. Horwitz, S., Reps, T., and Binkley, D., “Interprocedural slicing using dependence graphs,” *Proceedings of the SIGPLAN 88 Conference on Programming Language Design and Implementation,* (Atlanta, GA, June 22-24, 1988)*, ACM SIGPLAN Notices* **23**(7) pp. 35-46 (July 1988).

6. Horwitz, S., Prins, J., and Reps, T., “Integrating non-interfering versions of programs,” pp. 133-145 in *Conference Record of the 15th ACM Symposium on Principles of Programming Languages,* (San Diego, CA, January 13-15, 1988), ACM, New York (1988).

7. Horwitz, S., Prins, J., and Reps, T., “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).

8. Horwitz, S., Reps, T., and Binkley, D., “Interprocedural slicing using dependence graphs,” *ACM Transactions on Programming Languages and Systems* **12**(1) pp. 26-60 (January 1990).

9. Kuck, D. J., Muraoka, Y., and Chen, S. C., “On the number of operations simultaneously executable in FORTRAN-like programs and their resulting speed-up,” *IEEE Transactions on Computers C-21*, pp. 1293-1310 (December 1972).

10. Kuck, D.J., *The Structure of Computers and Computations, Vol. 1,* John Wiley and Sons, New York, NY (1978).

11. Kuck, D.J., Kuhn, R.H., Leasure, B., Padua, D.A., and Wolfe, M., “Dependence graphs and compiler optimizations,” pp. 207-218 in *Conference Record of the Eighth ACM Symposium on Principles of Programming Languages,* (Williamsburg, VA, January 26-28, 1981), ACM, New York (1981).

12. Ottenstein, K.J. and Ottenstein, L.M., “The program dependence graph in a software development environment,” *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, (Pittsburgh, PA, April 23-25, 1984), ACM SIGPLAN Notices* **19**(5) pp. 177-184 (May 1984).

13. Towle, R., “Control and data dependence for program transformations,” TR 76-788, Department of Computer Science, University of Illinois, Urbana-Champaign, IL (March 1976).