The Semantics of Program Slicing

THOMAS REPS and WUU YANG
University of Wisconsin – Madison

A slice of a program with respect to a program point p and variable x consists of all statements of the program that might affect the value of x at point p. Slices can be extracted particularly easily from a program representation called a dependence graph, originally introduced as an intermediate program representation for performing optimizing, vectorizing, and parallelizing transformations. Such slices are of a slightly restricted form: rather than permitting a program to be sliced with respect to program point p and an *arbitrary* variable, a slice must be taken with respect to a variable that is *defined* at or *used* at p.

This paper concerns the relationship between the execution behavior of a program and the execution behavior of its slices. Our main results are those stated as the Slicing Theorem and the Termination Theorem. The Slicing Theorem demonstrates that a slice captures a portion of a program's behavior in the sense that, for any initial state on which the program halts, the program and the slice compute the same sequence of values for each element of the slice. The Termination Theorem demonstrates that if a program is decomposed into (two or more) slices, the program halts on any state for which all the slices halt.

These results are used to provide semantic justification for a program-integration algorithm of Horwitz, Prins, and Reps.

Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Distribution and Maintenance – *enhancement, restructuring, version control*; D.2.9 [**Software Engineering**]: Management – *programming teams, software configuration management*; D.3.4 [**Programming Languages**]: Processors – *compilers, interpreters, optimization*; E.1 [**Data Structures**] *graphs*

General Terms: Theory

Additional Key Words and Phrases: control dependence, data dependence, data-flow analysis, dependence graph, program slice, program integration, semantics, termination

---

1. INTRODUCTION

The *slice* of a program with respect to program point p and variable x consists of all statements and predicates of the program that might affect the value of x at point p [16]. Program slicing can be used to isolate individual computation threads within a program, which can help a programmer understand complicated code. Program slicing is also used by the algorithm for automatically integrating program variants described in [5] and [12]; slices are used to compute a safe approximation to the computation threads that have changed between a program P and a modified version of P, and to help determine whether two different modifications to P interfere.

This work was supported in part by the National Science Foundation under grant DCR-8552602, 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, and Xerox.

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

The original algorithm given for program slicing was expressed as a sequence of data flow analysis problems [16]. An alternative (and more practical) approach was put forward in [10], where it was pointed out that the slice of a program with respect to an initial set of vertices could be computed by walking backwards over the edges of the *program dependence graphs* being proposed as the internal representation of programs in a program development environment. The kind of slicing that can be performed using a program dependence graph is, however, somewhat restricted: rather than permitting a program to be sliced with respect to program point *p* and an *arbitrary* variable, a slice must be taken with respect to a variable that is *defined* at or *used* at *p*. It is this restricted kind of slice that is studied here.

This paper presents results that characterize the relationship between the execution behavior of a program and the execution behavior of its slices. Our main results on slicing are those stated as the Slicing Theorem and the Termination Theorem. The Slicing Theorem demonstrates that a slice captures a portion of a program's behavior in the sense that, for any initial state on which the program halts, the program and the slice compute the same sequence of values for each element of the slice. The Termination Theorem demonstrates that if a program is decomposed into (two or more) slices, the program halts on any state for which all the slices halt.

These theorems are then used to provide semantic justification for the program-integration algorithm presented in [512]; the integration algorithm presented there either merges two program versions with a base version of the program or determines that the variants incorporate interfering changes. In the algorithm, slicing is used to determine which elements from the base program and its variants should be incorporated in the integrated program. The integrated program is created by (1) finding slices that represent the changed computation threads of the variant programs as well as the computation threads of the base program that are preserved in both variants, (2) combining these slices to form the merged graph, and (3) testing for interference by checking whether the slices that were combined to form the merged graph are preserved (as slices of the merged graph). (A prototype implementation of an integration tool based on this algorithm has been embedded in a program editor created using the Synthesizer Generator [1113].)

The Slicing and Termination Theorems are used to prove a theorem, the Integration Theorem, that characterizes the execution behavior of the integrated program in terms of the behaviors of the base program and the two variants; the Integration Theorem demonstrates that the integrated program produced by a successful integration preserves the changed behaviors of both variants as well as the behavior of the base program that is unchanged in both variants.

The rest of the paper is organized as follows: Section 2 defines program dependence graphs and the operation of slicing a program dependence graph. Section 3 presents the proof of the Feasibility Lemma. Section 4 presents the proof of the Slicing Theorem. Section 5 presents the proof of the Termination Theorem. In Section 6, the Slicing Theorem and the Termination Theorem are used to prove the Integration Theorem. Section 7 discusses the relation of the work described to previous work.

2. TERMINOLOGY AND NOTATION

We are concerned with a restricted programming language with the following characteristics: expressions contain only scalar variables and constants; statements are either assignment statements, conditional statements, while loops, or a restricted kind of "output statement" called an *end statement*, which can only appear at the end of a program. An end statement names one or more of the variables used in the program. Thus a program is of the form:

program *id*
stmt_list
end(id^*)

Our discussion of the language's semantics is in terms of the following informal model of execution. We assume a standard operational semantics for sequential execution of the corresponding flowchart (control flow graph): at any moment there is a single locus of control; the execution of each assignment statement or predicate passes control to a single successor; the execution of each assignment statement changes a global execution state. An execution of the program on some initial state also yields a (possibly infinite) sequence of values for each predicate and assignment statement in the program; the i^{th} element in the sequence for program element e consists of the value computed when e is executed for the i^{th} time. The variables named in the end statement are those whose final values are of interest to the programmer; when execution terminates, the final state is defined on only those variables in the end statement.

The abstract syntax of the language is defined as the terms of the types *id*, *exp*, *stmt*, *stmt_list*, and *program* constructed using the operators *Assign*, *While*, *IfThenElse*, *StmtList*, and *Program*. The five operators of the abstract syntax have the following definitions:

$Assign$: $id \times exp \rightarrow stmt$
$While$: $exp \times stmt\_list \rightarrow stmt$
$IfThenElse$: $exp \times stmt\_list \times stmt\_list \rightarrow stmt$
$StmtList$: $stmt^+ \rightarrow stmt\_list$
$Program$: $id \times stmt\_list \times id^* \rightarrow program$

In operator *Program*, the initial *id* represents the program name, and the id^* component represents the variables named in the end statement.

We also introduce an overloaded constant, *Null*, which is used to represent null trees of type *id*, *exp*, *stmt*, and *stmt_list*:

$Null$: $\rightarrow id$
$Null$: $\rightarrow exp$
$Null$: $\rightarrow stmt$
$Null$: $\rightarrow stmt\_list$

Null is introduced solely for technical reasons, and is never an element of a program's abstract syntax tree.

Henceforth, we use "program" and "abstract syntax tree" synonymously.

2.1. The Program Dependence Graph

Different definitions of program dependence representations have been given, depending on the intended application; they are all variations on a theme introduced in [7, 14, 8, 9], and share the common feature of having an explicit representation of data dependences (see below). The "program dependence graphs" defined in [3] introduced the additional feature of an explicit representation for control dependences (see below). Although the definition of program dependence graph given below covers only the restricted language described earlier, and hence is less general than the one given in [3], the structures we define share the feature of representing both control and data dependences and we will refer to them as "program dependence graphs," borrowing the term from [3].

The *program dependence graph* (or PDG) for a program P, denoted by G_P, is a directed graph whose vertices are connected by several kinds of edges.[1] The vertices of G_P represent the assignment statements and control predicates that occur in program P. In addition, G_P includes three other categories of vertices:

1) There is a distinguished vertex called the *entry vertex*.

2) For each variable x for which there is a path in the standard control-flow graph for P on which x is used before being defined (see [1]), there is a vertex called the *initial definition of x*. This vertex represents an assignment to x from the initial state. The vertex is labeled "$x := InitialState(x)$."

3) For each variable x named in P's end statement, there is a vertex called the *final use of x*. This vertex represents an access to the final value of x computed by P, and is labeled "$FinalUse(x)$."

The edges of G_P represent *dependences* between program components. An edge represents either a *control dependence* or a *data dependence*. Control dependence edges are labeled either **true** or **false**, and the source of a control dependence edge is always the entry vertex or a predicate vertex. A control dependence edge from vertex v_1 to vertex v_2, denoted by $v_1 \longrightarrow_c v_2$, means that during execution, whenever the predicate represented by v_1 is evaluated and its value matches the label on the edge to v_2, then the program component represented by v_2 will be executed (although perhaps not immediately). A method for determining control dependence edges for arbitrary programs is given in [3]; however, because we are assuming that programs include only assignment, conditional, and while statements, the control dependence edges of G_P can be determined in a much simpler fashion. For the language under consideration here, the control dependence edges reflect a program's nesting structure; a program dependence graph contains a *control dependence edge* from vertex v_1 to vertex v_2 of G_P iff one of the following holds:

i) v_1 is the entry vertex, and v_2 represents a component of P that is not subordinate to any control predicate; these edges are labeled **true**.

ii) v_1 represents a control predicate, and v_2 represents a component of P immediately subordinate to the control construct 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** branch, respectively.[2]

A data dependence edge from vertex v_1 to vertex v_2 means that the program's computation might be changed if the relative order of the components represented by v_1 and v_2 were reversed. In this paper,

---

[1] We make use of the following graph terminology:

1) A *directed graph G* consists of a set of *vertices* $V(G)$ and a set of *edges* $E(G)$, where $E(G) \subseteq V(G) \times V(G)$. Each edge $(b, c) \in E(G)$ is directed from b to c; we say that b is the *source* and c the *target* of the edge.

2) A *multi-graph* is a graph with a bag of edges (*i.e.* duplicate edges may exist in the graph).

3) Given directed graphs $A = (V_A, E_A)$ and $B = (V_B, E_B)$, that may or may not be disjoint, the *union* of A and B is defined as:

$A \cup B = (V_A \cup V_B, E_A \cup E_B)$

4) Given a directed graph $A = (V, E)$, a *path* from vertex a to vertex b is a sequence of vertices, $[v_1, v_2, ..., v_k]$, such that: $a = v_1$, $b = v_k$, and $\{ (v_i, v_{i+1}) \mid i = 1, ..., k-1 \} \subseteq E$.

5) Given a directed graph $A = (V, E)$ and a set of vertices $V' \subseteq V$, the *projection* of A onto V' is the graph (V', E'), where $E' = \{(v, w) \mid v, w \in V'$ and there exists a path $[v = v_1, v_2, ..., v_k = w]$ such that $v_2, ..., v_{k-1} \notin V'\}$. (That is, the projection of A onto V' has an edge from $v \in V'$ to $w \in V'$ when there exists a path from v to w in A that does not pass through any other elements of V'.)

[2] In other definitions that have been given for control dependence edges, there is an additional edge for each predicate of a **while** statement − each predicate has an edge 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) when the predicate evaluates to **true**. This kind of edge turns out to be unnecessary for our purposes and hence is left out of our definition.

program dependence graphs contain two kinds of data-dependence edges, representing *flow dependences* and *def-order dependences*.

A program dependence graph contains a flow dependence edge from vertex v_1 to vertex v_2 iff all of the following hold:

i) v_1 is a vertex that defines variable x.

ii) v_2 is a vertex that uses x.

iii) Control can reach v_2 after v_1 via an execution path along which there is no intervening definition of x. That is, there is a path in the standard control-flow graph for the program [1] by which the definition of x at v_1 reaches the use of x at v_2. (Initial definitions of variables are considered to occur at the beginning of the control-flow graph, and final uses of variables are considered to occur at its end.)

A flow dependence that exists from vertex v_1 to vertex v_2 will be denoted by $v_1 \longrightarrow_f v_2$.

Flow dependences are further classified as *loop independent* or *loop carried*. 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 i), ii), and iii) above, the following also hold:

iv) There is an execution path that both satisfies the conditions of iii) above and includes a backedge to the predicate of loop L; and

v) 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 i), ii), and iii) above, there is an execution path that satisfies iii) 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 program dependence graph contains a def-order dependence edge from vertex v_1 to vertex v_2 iff all of the following hold:

i) v_1 and v_2 both define the same variable.

ii) v_1 and v_2 are in the same branch of any conditional statement that encloses both of them.

iii) There exists a program component v_3 such that $v_1 \longrightarrow_f v_3$ and $v_2 \longrightarrow_f v_3$.

iv) v_1 occurs to the left of v_2 in the program's abstract syntax tree.

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

Note that a program dependence graph 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 between two vertices, each is labeled by a vertex that is flow-dependent on both the definition that occurs at the edge's source and the definition that occurs at the edge's target.

Example. Figure 1 shows an example program and its program dependence graph. The boldface arrows represent control dependence edges; dashed arrows represent def-order dependence edges; solid arrows represent loop-independent flow dependence edges; solid arrows with a hash mark represent loop-carried flow dependence edges.

The data-dependence edges of a program dependence graph are computed using data-flow analysis. For the restricted language considered in this paper, the necessary computations can be defined in a syntax-directed manner (see [4]).

```
program Main
   sum := 0;
   x := 1;
   while x < 11 do
      sum := sum + x;
      x := x + 1
   od
end(x, sum)
```

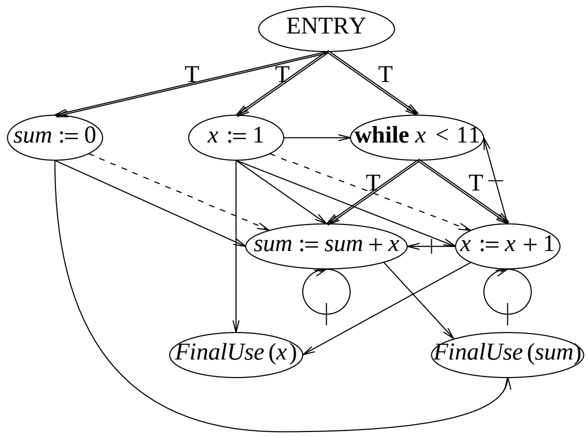

Figure 1. An example program, which sums the integers from 1 to 10 and leaves the result in the variable *sum*, and its program dependence graph. The boldface arrows represent control dependence edges, dashed arrows represent def-order dependence edges, solid arrows represent loop-independent flow dependence edges, and solid arrows with a hash mark represent loop-carried flow dependence edges.

The relationship between a program's PDG and the program's execution behavior has been addressed in [6]. In particular, it is shown in [6] that if the PDGs of two programs are isomorphic then the programs have the same behavior. The concept of "programs with the same behavior" is formalized as the concept of *strong equivalence*, defined as follows:

DEFINITION. Two programs P and Q are *strongly equivalent* iff for any state σ, either P and Q both diverge when initiated on σ or they both halt with the same final values for all variables. If P and Q are not strongly equivalent, we say they are *inequivalent*.

The term "divergence" refers to both non-termination (for example, because of infinite loops) and abnormal termination (for example, because of division by zero or the use of an out-of-bounds array index).

The main result of [6] is the following theorem: (the symbol $\approx$ denotes isomorphism between program dependence graphs).

THEOREM. (EQUIVALENCE THEOREM). *If P and Q are programs for which $G_P \approx G_Q$, then P and Q are strongly equivalent*.

Restated in the contrapositive the theorem reads: inequivalent programs have non-isomorphic program dependence graphs. (We prove a stronger form of this theorem in Section 4.2.)

2.2. Program Slices

For a vertex s of a PDG G, the *slice* of G with respect to s, written as G/s, is a graph containing all vertices on which s has a transitive flow or control dependence (*i.e.* all vertices that can reach s via flow or control edges): $V(G/s) = \{ w \mid w \in V(G) \quad w \longrightarrow^*_{c,f} s \}$. We extend the definition to a set of vertices $S = \bigcup_i s_i$ as follows: $V(G/S) = V(G/(\bigcup_i s_i)) = \bigcup_i V(G/s_i)$. It is useful to define $V(G/v) = \emptyset$ for any $v \notin G$.

The edges in the graph G/S are essentially those in the subgraph of G induced by $V(G/S)$, with the exception that a def-order edge $v \longrightarrow_{do(u)} w$ is only included if, in addition to v and w, $V(G/S)$ also contains the vertex u that is directly flow dependent on the definitions at v and w. In terms of the three types of edges in a PDG we have:

$$
\begin{aligned}
E(G/S) = \quad & \{ (v \longrightarrow_f w) \mid (v \longrightarrow_f w) \in E(G) \ \ v, w \in V(G/S) \} \\
& \cup \{ (v \longrightarrow_c w) \mid (v \longrightarrow_c w) \in E(G) \ \ v, w \in V(G/S) \} \\
& \cup \{ (v \longrightarrow_{do(u)} w) \mid (v \longrightarrow_{do(u)} w) \in E(G) \ \ u, v, w \in V(G/S) \}
\end{aligned}
$$

Example. Figure 2 shows the graph resulting from taking a slice of the program dependence graph from Figure 1 with respect to the final-use vertex for x.

The following lemma demonstrates a useful property of program slicing.

LEMMA. (Decomposition Lemma). *For any collection* $\bigcup_i s_i$ *of program points, we have* $\bigcup_i (G/s_i) = G/\bigcup_i s_i$.

PROOF. The graph $\bigcup_i G/s_i$ consists of vertices $\bigcup_i V(G/s_i)$ and edges $\bigcup_i E(G/s_i)$.

1) By the definition of the vertex set of a slice, $\bigcup_i V(G/s_i) = V(G/\bigcup_i s_i)$.

2) (a) For each edge $u \longrightarrow w \in \bigcup_i E(G/s_i)$ we have $u \longrightarrow w \in E(G/s_i)$ for some i. Because $\{ s_i \} \subseteq \bigcup_i s_i$, $u \longrightarrow w \in E(G/\bigcup_i s_i)$, so $\bigcup_i E(G/s_i) \subseteq E(G/\bigcup_i s_i)$.

(b) For each control or flow edge $u \longrightarrow_{c,f} w \in E(G/\bigcup_i s_i)$ we have $w \in V(G/\bigcup_i s_i) = \bigcup_i V(G/s_i)$. Hence $w \in V(G/s_i)$ for some i. Since $u \longrightarrow_{c,f} w$ we must have $u \in V(G/s_i)$ as well, so

---

```
program Main
   x := 1;
   while x < 11 do
      x := x + 1
   od
end(x)
```

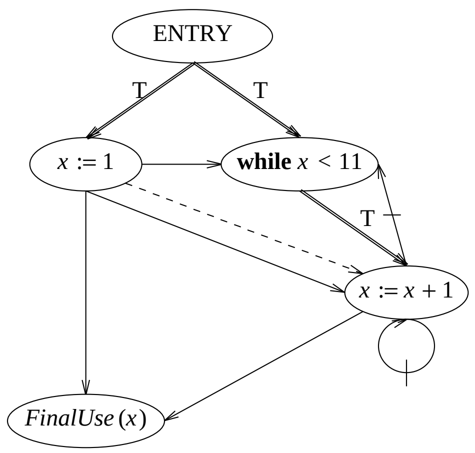

Figure 2. The graph that results from slicing the example from Figure 1 with respect to the final-use vertex for x.

$u \longrightarrow_{c,f} w \in E(G/s_i) \subseteq \bigcup_i E(G/s_i)$.

By the definition of the edge set of a slice, for each def-order edge $u \longrightarrow_{do(t)} w \in E(G/\bigcup_i s_i)$ we have $t \in V(G/\bigcup_i s_i) = \bigcup_i V(G/s_i)$. Hence $t \in V(G/s_i)$ for some i. Since $u \longrightarrow_f t$ and $w \longrightarrow_f t$ we have $t,u,w \in V(G/s_i)$, so $u \longrightarrow_{do(t)} w \in E(G/s_i) \subseteq \bigcup_i E(G/s_i)$.

Since G contains only flow, control and def-order edges, we have $E(G/\bigcup_i s_i) \subseteq \bigcup_i E(G/s_i)$.

Combining (a) and (b) we have $\bigcup_i E(G/s_i) = E(G/\bigcup_i s_i)$.

Combining (1) and (2) we have $\bigcup_i (G/s_i) = G/\bigcup_i s_i$. □

3. THE FEASIBILITY LEMMA

Our first result concerns a syntactic property of slices: we say that a graph G is a *feasible program dependence graph* iff G is (isomorphic to[3]) the program dependence graph of some program P (*i.e.* $G \approx G_P$). We now show that for any program P and vertex set S, the slice G_P/S is also a feasible PDG; the proof proceeds by showing that G_P/S corresponds to the program obtained by restricting the syntax tree of P to just the statements and predicates in $V(G_P/S)$.

LEMMA. (FEASIBILITY LEMMA). *For any program P, if G_Q is a slice of G_P (with respect to some set of vertices), then G_Q is a feasible PDG.*

PROOF. We construct a new program Q' from P and G_Q as follows: the elements (statements and predicates) of Q' correspond to the vertices of G_Q; each element of Q' is subordinate to the same element that it is subordinate to in P; the elements of Q' occur in the same relative order as they do in P. The variables that appear in the end statement of Q' are those variables whose final-use vertices are in G_Q. We use $G_{Q'}$ to denote the program dependence graph of Q'.

Because each element of Q' is subordinate to the same element that it is subordinate to in P, and because elements of Q' occur in the same order as they occur in P, the control flow graph for program Q' is the projection of the control flow graph for program P onto the elements of Q'. That is, if a and b are elements of Q', the projection of any path from a to b in the control flow graph of P onto the set of elements of Q' is a path in the control flow graph of Q'. Furthermore, every path from a to b in the control flow graph of Q' is the projection of some path from a to b (and possibly several such paths) in the control flow graph of P.

>From the construction of Q', the only possible differences between the vertex sets of G_Q and $G_{Q'}$ is in their initial-definition vertices. By the definition of the vertex set of a slice, if there is an initial-definition vertex a for variable x in $V(G_Q)$, there must be a flow edge $a \longrightarrow_f b$ in $E(G_Q)$, where b is not an initial-def vertex. Since $a \longrightarrow_f b \in E(G_Q)$, it must be that $a \longrightarrow_f b \in E(G_P)$. This means that there is a path from the beginning of the control flow graph of P to b that is free of any definition to x. The projection of this path onto the elements of Q' is a path in Q' from the beginning of the control flow graph of Q' to b and contains no definition to x. Consequently, $V(G_{Q'})$ must contain an initial-definition vertex for x, which corresponds to vertex a in $V(G_Q)$.

---

[3]For brevity, when two graphs are related by some mapping (for example, an isomorphism or an embedding) we frequently blur the distinction between corresponding elements of the two graphs; strictly speaking, we should refer to "elements that correspond under the mapping."

Arguing in the other direction, suppose there is an initial-definition vertex a for variable x in $V(G_{Q'})$ and a flow edge $a \longrightarrow_f b$ that occurs in $E(G_{Q'})$ but not in $E(G_Q)$. Since $b \in V(G_Q)$ but $a \longrightarrow_f b \notin E(G_Q)$, by the definition of the edge set of a slice, $a \longrightarrow_f b$ cannot be in $E(G_P)$. Therefore, along each path from the entry vertex to b in P there must be a redefinition of x. Along each such path p, let c_p be the last redefinition site. Since $c_p \longrightarrow_f b$ is in $E(G_P)$ and b is in $V(G_Q)$, $c_p \longrightarrow_f b$ is in $E(G_Q)$; the vertex c_p itself must be in $V(G_Q)$ and hence in Q'. Because every path from the entry vertex to b in the control flow graph of Q' is a projection of a path p from the entry vertex to b in the control flow graph of P, there must be a redefinition of x on each path from the entry vertex to b in Q'. This means that $a \longrightarrow_f b$ cannot be in $E(Q')$, which is a contradiction, hence there does exist a flow edge $a \longrightarrow_f b$ in $E(G_P)$ where a is the initial-definition vertex for x in P. Because $b \in V(G_{Q'})$ and b itself is not an initial-definition vertex, by the construction of Q' it must be that $b \in V(G_Q)$. Consequently, by the definition of the vertex set of a slice, $a \in V(G_Q)$ and $a \longrightarrow_f b \in E(G_Q)$.

We have shown above that G_Q and $G_{Q'}$ have isomorphic vertex sets, what remains to be shown is that G_Q and $G_{Q'}$ have isomorphic edge sets.

Sub-proof 1. If the edge $(a, b) \in E(G_Q)$, then $(a, b) \in E(G_{Q'})$.

1) By the definition of the edge set of a slice, if $a \longrightarrow_c b$ is a control edge in $E(G_Q)$, then $a \longrightarrow_c b$ is a control edge in $E(G_P)$, which means that b is subordinate to a in program P. Because an element in program Q' is subordinate to the same element that it is subordinate to in P, $a \longrightarrow_c b$ is in $E(G_{Q'})$, as well.

2) By the definition of the edge set of a slice, if $a \longrightarrow_f b$ is a flow edge in $E(G_Q)$, then $a \longrightarrow_f b$ is in $E(G_P)$, which means that there is a path in the control flow graph of P from a to b without any redefinition to the target variable of a. The projection of this path onto the elements of Q' is a path in Q' that contains no redefinition to the target variable of a; thus, $a \longrightarrow_f b$ is in $E(G_{Q'})$.

3) By the definition of the edge set of a slice, if $a \longrightarrow_{do(c)} b$ is a def-order edge in $E(G_Q)$, then there are flow edges $a \longrightarrow_f c$ and $b \longrightarrow_f c$ in $E(G_Q)$. >From the argument in (2), the edges $a \longrightarrow_f c$ and $b \longrightarrow_f c$ also occur in $E(G_{Q'})$. Because a occurs to the left of b in P's abstract syntax tree, a occurs to the left of b in the abstract syntax tree of Q'. Therefore, $a \longrightarrow_{do(c)} b$ is in $E(G_{Q'})$.

Sub-proof 2. If edge $(a, b) \in E(G_{Q'})$, then $(a, b) \in E(G_Q)$.

1) If $a \longrightarrow_c b$ is a control edge in $E(G_{Q'})$, then b is subordinate to a in Q', hence b is subordinate to a in P. Therefore, $a \longrightarrow_c b$ is a control edge in $E(G_P)$ and, by the definition of the edge set of a slice, the control edge $a \longrightarrow_c b$ is a member of $E(G_Q)$.

2) Suppose $a \longrightarrow_f b$ is a flow edge that occurs in $E(G_{Q'})$ but not in $E(G_Q)$. Since $a, b \in V(G_Q)$ but $a \longrightarrow_f b \notin E(G_Q)$, by the definition of the edge set of a slice, $a \longrightarrow_f b$ cannot be in $E(G_P)$. Therefore, along each path from a to b in P there must be a redefinition of the target variable of a.

   Along each such path p, let c_p be the last redefinition site. Since $c_p \longrightarrow_f b$ is in $E(G_P)$ and b is in $V(G_Q)$, $c_p \longrightarrow_f b$ is in $E(G_Q)$; the vertex c_p itself must be in $V(G_Q)$ and hence in Q'. Because every path from a to b in the control flow graph of Q' is a projection of a path p from a to b in the control flow graph of P, there must be a redefinition of the target variable of a on each path from a to b in Q'. This means that $a \longrightarrow_f b$ cannot be in $E(Q')$, which is a contradiction; therefore, $a \longrightarrow_f b$ is a flow edge in $E(G_Q)$.

3) If $a \longrightarrow_{do(c)} b$ is a def-order edge in $E(G_{Q'})$, then $a \longrightarrow_f c$ and $b \longrightarrow_f c$ are in $E(G_{Q'})$. >From the argument in (2), the edges $a \longrightarrow_f c$ and $b \longrightarrow_f c$ are in $E(G_P)$ and $E(G_Q)$, as well. Because $a \longrightarrow_{do(c)} b$ is in $E(G_{Q'})$ a occurs to the left of b in Q', hence a occurs to the left of b in P and

therefore $a \longrightarrow_{do(c)} b$ is in $E(G_P)$. Now a, b, and c are all in $V(G_Q)$; thus, by the definition of the edge set of a slice, $a \longrightarrow_{do(c)} b$ is in $E(G_Q)$.

We have shown that G_Q is isomorphic to $G_{Q'}$ and hence corresponds to program Q'. Therefore, the slice of a feasible PDG is always a feasible PDG. □

Note that there may be programs other than Q' whose program dependence graph is isomorphic to G_Q. By the Equivalence Theorem, all such programs are strongly equivalent to Q' [6].

Since a slice of a feasible program dependence graph is feasible, and programs with isomorphic program dependence graphs are strongly equivalent, we can speak of "a slice of a program" as well as "a slice of a program dependence graph." We say program Q is a slice of program P with respect to a set of program points, S, when $G_Q \approx (G_P / S)$, and write this as P / S.

4. THE SEMANTICS OF PROGRAM SLICING

We now turn to the relationship between the execution behaviors of a program and a slice of the program. Because of the way a program slice is derived from a program, it is not unreasonable to expect that the program and the slice exhibit similar execution behavior. However, because a diverging computation may be "sliced out," a program and a slice of the program do not necessarily exhibit identical execution behaviors; in particular, a slice may produce a result on some initial states for which the original program diverges. For example, the program shown below on the left always diverges, whereas the program on the right, obtained by slicing the left-hand-side program with respect to variable x at the program's **end** statement, always converges:

```
program Main                    program Main
   x := 1;                         x := 0
   while true do                end(x)
      x := x + 1
   od;
   x := 0
end(x)
```

The main result of this section is the following theorem, which asserts that a slice captures a portion of the program's behavior in the sense that, for any initial state on which the program halts, the program and the slice compute the same sequence of values for each element of the slice. (In our case a program point may be (1) an assignment statement, (2) a control predicate, or (3) a final use of a variable in an end statement. By "computing the same sequence of values" at each corresponding point we mean: (1) for any assignment statement the same *sequence* of values are assigned to the target variable; (2) for a predicate the same *sequence* of boolean values are produced; and (3) for each final use the same value for the variable is produced.)

THEOREM. (SLICING THEOREM). *Let Q be a slice of program P with respect to a set of vertices. If σ is a state on which P halts, then for any state σ' that agrees with σ on all variables for which there are initial-definition vertices in G_Q: (1) Q halts on σ', (2) P and Q compute the same sequence of values at each program point of Q, and (3) the final states agree on all variables for which there are final-use vertices in G_Q.*

(The third clause of the theorem's conclusion is implied by the second clause; it is stated explicitly to emphasize what the theorem says about programs viewed as state-transformers.)

The proof of the Slicing Theorem relies on a lemma, the Subtree Slicing Lemma, which is stated and proven in Section 4.3.

4.1. Additional Terminology and Results

The subgraph induced by the *control* dependences of program dependence graph G_P forms a tree that is closely related to the abstract syntax tree for program *P*. The control dependence subtree is rooted at the entry vertex of G_P, which corresponds to the Program node at the root of *P*'s abstract syntax tree. Each predicate vertex *v* of G_P corresponds to an interior node of the abstract syntax tree; the node is a While node or an IfThenElse node depending on whether *v* is labeled with **while** or **if**, respectively. Each assignment vertex of G_P corresponds to an Assign node of the abstract syntax tree.

The control dependence subtree rooted at a vertex *v* of G_P corresponds to the subtree of the abstract syntax tree that is rooted at the control construct that corresponds to *v*. Because of this correspondence, for brevity we use phrases, such as "the flow edges whose source is in subtree *T*," which are, strictly speaking, not correct when *T* is a subtree of the abstract syntax tree. What "*T*" refers to is the subgraph induced by *T* in G_P's control dependence subgraph.

Imported and exported variables

Our goal is to show that a slice of a program exhibits a portion of the program's behavior in the sense that they are equivalent state transformers with respect to certain variables. In making this argument, it is necessary to discuss the state-transforming properties of subtrees. The state-transforming properties of a subtree are characterized in terms of the subtree's *imported* and *exported* variables.

DEFINITION. The *outgoing flow edges* of a subtree *T* consist of all the loop-independent flow edges whose source is in *T* but whose target is not in *T*, together with all the loop-carried flow edges for which the source is in *T* and the edge is carried by a loop that encloses *T*. Note that the target of an outgoing loop-carried flow edge may or may not be in *T*. The variables *exported* from a subtree *T* are the variables defined at the source of an outgoing flow edge.

DEFINITION. The *incoming flow edges* of a subtree *T* consist of all the loop-independent flow edges whose target is in *T* but whose source is not in *T*, together with all the loop-carried flow edges for which the target is in *T* and the edge is carried by a loop that encloses *T*. Note that the source of an incoming loop-carried flow edge may or may not be in *T*. The *incoming def-order edges* of a subtree *T* consist of all the def-order edges whose target is in *T* but whose source is not in *T*. The variables *imported* by a subtree *T* are the variables defined at the source of an incoming flow edge or at the source of an incoming def-order edge.

Note that there are loop-independent flow edges to all final-use vertices of a program dependence graph; thus, the exported variables of a program *P* consist of all the variables that occur in *P*'s **end** statement. Similarly, there are loop-independent flow edges from all of the initial-definition vertices; thus, the imported variables of a program *P* consist of those variables that may get their values from the initial state.

The Self-Equivalence Lemma

The Self-Equivalence Lemma, proved in [6], shows that the definitions of imported and exported variables are consistent with each other and can be used to characterize the state-transforming properties of a subtree.

LEMMA. (SELF-EQUIVALENCE LEMMA). *Let T be a subtree of program P. Then T is strongly equivalent to T relative to T's imported and exported variables (as defined in the context given by P).*

Corresponding subtrees

Let Q be a slice of P with respect to a set of program points. There is natural correspondence between subtrees in P and subtrees in Q, defined as follows:

DEFINITION. Let Q be a slice of P with respect to some set of program points. For each subtree U of Q with root u, U *corresponds* to the subtree of P whose root is u. For each subtree T of P, if the root t of T occurs in Q, T *corresponds* to the subtree of Q rooted at t; if t does not occur in Q, T *corresponds* to the tree *Null*.

Thus, for each subtree of Q, there is always a corresponding subtree of P, and *vice versa*, although for a subtree of P the corresponding subtree of Q may be the tree *Null*.

Note that the "corresponds to" relation respects the hierarchical structure of programs: children of roots of corresponding subtrees are the roots of corresponding subtrees.

4.2. A Strong Form of the Equivalence Theorem

The Equivalence Theorem, which states that programs with isomorphic program dependence graphs are strongly equivalent with respect to the imported and exported variables, was proven in [6]. To prove the Slicing Theorem, we need a stronger form of the Equivalence Theorem, which states that, when initiated on the same state, programs with isomorphic program dependence graphs are not only strongly equivalent but actually compute the same sequence of values at each corresponding program point.

In [6] the Equivalence Theorem follows as a corollary of the following lemma:

LEMMA. (EQUIVALENCE LEMMA). *Suppose that P and Q are programs for which $G_P \approx G_Q$. Then for any subtrees T in P and U in Q that correspond, T and U are strongly equivalent relative to their imported and exported variables*.

In this section, we first prove the Subtree Equivalence Lemma, which is a strong form of the Equivalence Lemma; it states that for two programs with isomorphic program dependence graphs their corresponding subtrees compute the same sequence of values at each corresponding program point when they both terminate on a state. The strong form of the Equivalence Theorem then follows as a corollary of the Subtree Equivalence Lemma.

4.2.1. The Subtree Equivalence Lemma

LEMMA. (SUBTREE EQUIVALENCE LEMMA). *Suppose that P and Q are programs for which $G_P \approx G_Q$. Let T be a subtree of P and U be the corresponding subtree of Q. If σ is a state on which T halts, then for any state σ' that agrees with σ on their imported variables, (1) U halts on σ', (2) T and U compute the same sequence of values at each corresponding program point, and (3) the final states agree on their exported variables*.

Note that corresponding subtrees of P and Q have isomorphic program dependence graphs and the same imported and exported variables because P and Q have isomorphic program dependence graphs.

PROOF. By the Equivalence Lemma, T and U are strongly equivalent relative to their imported and exported variables, *i. e.*, when the executions of T and U are initiated on σ and σ', respectively, which agree on the imported variables, either they both diverge or they both halts with the same final values for all exported variables. Thus, (1) and (3) are simply a restatement of the Equivalence Lemma. We need to prove (2).

The proof is by structural induction on the abstract syntax of the programming language. The proof splits into five cases based on the abstract-syntax operator that appears at the root of T.

Throughout the proof, we use *Imp* and *Exp* to denote the imported and exported variables of T, respectively (T and U have the same imported and exported variables); we use σ_1 and σ_1' to denote states that agree on the imported variables, *Imp*. We use σ_i to denote a sequence of states in the execution of T initiated on σ_1, and we use σ_i' to denote the corresponding sequence of states in the execution of U initiated on σ_1'.

Case 1. The operator at the root of T is the Assign operator. Note that $T = U$ in this case and that T assigns to variable x as a function of variables $\{y_j\}$; *Imp* is either $\{y_j\}$ or $\{y_j\} \cup \{x\}$ (*Imp* is $\{y_j\} \cup \{x\}$ when T is the target of a def-order edge). Since the value of the *exp* is a function of $\{y_j\}$; and $\{y_j\} \subseteq Imp$, evaluating *exp* in both σ_1 and σ_1' yields the same value because they agree on *Imp*. Thus, T and U compute the same (sequence of) values.

Case 2. The operator at the root of T is the While operator. Since T halts, we may assume the execution of T halts after the j^{th} iteration, for some j. It is sufficient to show that (1) U also halts after the j^{th} iteration, and (2) in each iteration, T and U compute the same sequence of values at each corresponding program point.

We use Imp_{exp} and Exp_{exp} to denote the imported and exported variables of the *exp* component, respectively; we use Imp_{stmt_list} and Exp_{stmt_list} to denote the imported and exported variables of the *stmt_list* component, respectively. We use σ_i and σ_i' to denote the execution states before executing the i^{th} iterations of the loops of T and U starting from two states that agree on *Imp*, σ_1 and σ_1', respectively.

Because for a loop $Exp \subseteq Imp$,[4] it suffices to show that if σ_i and σ_i' agree on *Imp* then either T and U both halt in the states σ_i and σ_i', respectively, or else the i^{th} iterations compute the same sequence of values at each corresponding program point and result in the states σ_{i+1} and σ_{i+1}' that agree on *Imp*.

First, we show that $Imp = Imp_{exp} \cup Imp_{stmt\_list}$. It is clear that we could have written this with $\subseteq$, noting that Imp_{stmt_list} can include a variable x that is used at the target t of a loop-carried flow dependence edge where the dependence is carried by the loop. However, there then has to exist an incoming loop-independent flow edge to t, which implies that $v \in Imp$.

Let σ_i and σ_i' be states that agree on *Imp*. Therefore they agree on Imp_{exp}. Evaluating the condition (the *exp* component) in σ_i and σ_i' yields the same value. Hence, T and U compute the same (sequence of) values at the control predicate of the loop in the i^{th} iteration. If the condition evaluates to false, then both executions terminate in the states σ_i and σ_i', respectively.

Now suppose the condition evaluates to true. Let σ_i and σ_i' be states that agree on *Imp*; therefore they agree on Imp_{stmt_list}. Now T_{stmt_list} and U_{stmt_list} are corresponding subtrees. By the induction hypothesis of the structural induction, T_{stmt_list} and U_{stmt_list} compute the same sequence of values at each corresponding program point of the *stmt_list* in the i^{th} iteration and the final states, σ_{i+1} and σ_{i+1}', agree on Exp_{stmt_list}. We need to show that σ_{i+1} and σ_{i+1}' agree on *Imp*. If σ_{i+1} and σ_{i+1}' do not also agree on *Imp*, then let $x \in Imp$ be a variable on which they disagree (so $x \notin Exp_{stmt\_list}$). Now, by assumption, σ_i and σ_i' agree on *Imp*; therefore, at least one of the two executions of T_{stmt_list} and U_{stmt_list}, respectively, executed an assignment statement a that assigned a value to x and reached the end of the *stmt_list*. There are two cases to consider:

---

[4]If $x \in Exp_U$, then U contains an assignment a to x with an outgoing flow edge $a \longrightarrow_f b$. Because the loop may execute zero times, the assignment to x must be the target of a def-order edge $\ldots \longrightarrow_{do(b)} a$, hence $x \in Imp_U$.

(1) One possibility is that $x \in Imp$ because x is used in a condition or statement b that is the target of an incoming flow edge *If this were the case, then there must be an outgoing loop-carried flow edge* $a \longrightarrow_{lc(T)} b$ *or* $a \longrightarrow_{lc(U)} b$*, depending on whether* T_{stmt_list} *or* U_{stmt_list} *executed a. However, in either case,* $x \in Exp_{stmt\_list}$*, which contradicts our previous assumption.*

(2) The other possibility is that $x \in Imp$ because there is an incoming def-order edge $\ldots \longrightarrow_{do(c)} d$ in the *stmt_list*. However, this implies that there is an outgoing flow edge $a \longrightarrow_f c$ from T_{stmt_list} or U_{stmt_list}, depending on whether T_{stmt_list} or U_{stmt_list} executed a. In either case, however, $x \in Exp_{stmt\_list}$, which contradicts our previous assumption.

We conclude that σ_{i+1} and σ_{i+1}' agree on *Imp*. Therefore, U halts after exactly the j^{th} iteration and during each iteration T and U compute the same sequence of values at each corresponding program point.

Case 3. The operator at the root of T is the IfThenElse operator. Note that T and U have the same *exp* component. Because σ_1 and σ_1' agree on *Imp* and $Imp_{exp} \subseteq Imp$, evaluating the condition (the *exp* component) in σ_1 and σ_1' yields the same value; thus, T and U compute the same (sequence of) values at the control predicate of the IfThenElse statement. Without loss of generality, assume that the condition evaluates to true.

We use T_{true}, T_{false}, U_{true} and U_{false} to denote the respective branches of T and U. Note that T_{true} and U_{true} are corresponding subtrees and T_{false} and U_{false} are corresponding subtrees. We use Imp_{true} and Exp_{true} to denote the imported and exported variables of T_{true}, respectively; we use Imp_{false} and Exp_{false} to denote the imported and exported variables of T_{false}, respectively.

When execution is initiated in state σ_1, T terminates in σ_2; consequently T_{true} also terminates in σ_2. Since σ_1 and σ_1' agree on *Imp* and $Imp_{true} \subseteq Imp$, σ_1 and σ_1' agree on Imp_{true}. Because T_{true} and U_{true} are corresponding subtrees, the induction hypothesis tell us that, when execution is initiated in state σ_1', (1) U_{true} terminates in state σ_2' (hence U terminates in state σ_2'), and (2) T_{true} and U_{true} compute the same sequence of values at each corresponding program point of the true branch (hence T and U compute the same sequence of values at each corresponding program point.)

Case 4. The operator at the root of T is the StmtList operator. Let $T_1, T_2, \cdots, T_n$ denote the immediate subtrees of T. Note that all loop-independent flow edges and def-order edges from one subtree to another go from left to right; that is, if there is a loop-independent flow edge or a def-order edge from a vertex in a subtree T_i to a vertex in a different subtree T_j then $i < j$. Let $U_1, U_2, \cdots, U_n$ denote the immediate subtrees of U in the order as they occur in program Q. Each T_i corresponds to some subtree $U_{\pi(i)}$ that is an immediate subtree of U, and *vice versa*, where the mapping π is is a permutation over the interval $1..n$. Let π^{-1} denote the inverse permutation of π.

We use σ_i and $\sigma_{\pi(i)}'$ to denote the execution states before executing T_i and $U_{\pi(i)}$, respectively; we use Imp_i and Exp_i to denote the imported and exported variables, respectively, of T_i (hence of $U_{\pi(i)}$). By the Equivalence Lemma, T_i is strongly equivalent to $U_{\pi(i)}$ relative to Imp_i and Exp_i.

The proof of this case is by induction over i. We want to show that for all i, $1 \le i \le n$, if σ_1 and σ_1' agree on *Imp* and T halts on σ_1, then σ_i and $\sigma_{\pi(i)}'$ agree on Imp_i and T_i and $U_{\pi(i)}$ compute the same sequence of values at each corresponding program point. Note that, by the induction hypothesis of the structural induction, if σ_i and $\sigma_{\pi(i)}'$ agree on Imp_i then T_i and $U_{\pi(i)}$ either both diverge or both halt and compute the same sequence of values at each corresponding program point. Thus, we will concentrate on proving that σ_i and $\sigma_{\pi(i)}'$ agree on Imp_i, for all i, $1 \le i \le n$,

Base case. $i = 1$. First we show that σ_1' and $\sigma_{\pi(1)}'$ agree on Imp_1. (Note that Imp_1 is the set of the imported variables of T_1 and hence of $U_{\pi(1)}$.) If σ_1' and $\sigma_{\pi(1)}'$ do not agree on Imp_1, let $x \in Imp_1$ be a variable on which they disagree. The execution of the initial subsequence $U_1, U_2, \cdots, U_{\pi(1)-1}$ executed an assignment statement to x and reached the beginning of $U_{\pi(1)}$. Let the assignment statement a be in U_k. Since x is an imported variable of $U_{\pi(1)}$, $U_{\pi(1)}$ has an incoming loop-independent flow edge or an incoming def-order edge $a \longrightarrow \ldots$, whose source is in U_k. Since T and U have isomorphic program dependence graphs, there is a corresponding loop-independent flow edge or a corresponding def-order edge from a vertex in $T_{\pi^{-1}(k)}$ to a vertex in T_1. Therefore, $\pi^{-1}(k) < 1$, which is a contradiction because T_1 is the first immediate subtree of T. We conclude that σ_1' and $\sigma_{\pi(1)}'$ agree on Imp_1.

Because σ_1 and σ_1' agree on Imp and $Imp_1 \subseteq Imp$, σ_1 and σ_1' agree on Imp_1. Because σ_1 and σ_1' agree on Imp_1 and σ_1' and $\sigma_{\pi(1)}'$ agree on Imp_1, σ_1 and $\sigma_{\pi(1)}'$ agree on Imp_1. By the induction hypothesis of the structural induction, T_1 and $U_{\pi(1)}$ compute the same sequence of values at each corresponding program point on σ_1 and $\sigma_{\pi(1)}'$, respectively.

Induction step. The induction hypothesis is: if σ_1 and σ_1' agree on Imp and T halts on σ_1, then σ_j and $\sigma_{\pi(j)}'$ agree on Imp_j and T_j and $U_{\pi(j)}$ compute the same sequence of values at each corresponding program point, for $1 \le j \le i$. Thus, if $\hat{\sigma}_1$ and $\hat{\sigma}_1'$ are arbitrary states that agree on Imp and T halts on $\hat{\sigma}_1$, we need to show that $\hat{\sigma}_{i+1}$ and $\hat{\sigma}_{\pi(i+1)}'$ agree on Imp_{i+1} and T_{i+1} and $U_{\pi(i+1)}$ compute the same sequence of values at each corresponding program point.

First we show that $\hat{\sigma}_{i+1}$ and $\hat{\sigma}_{\pi(i+1)}'$ agree on Imp_{i+1}. (Note that Imp_{i+1} is the set of the imported variables of T_{i+1} and hence of $U_{\pi(i+1)}$.) If $\hat{\sigma}_{i+1}$ and $\hat{\sigma}_{\pi(i+1)}'$ do not agree on Imp_{i+1}, let $x \in Imp_{i+1}$ be a variable on which they disagree. There are now two cases to consider:

(1) If there is no assignment statement to x in the initial subsequence $T_1, T_2, \cdots, T_i$, then $\hat{\sigma}_1$ and $\hat{\sigma}_{i+1}$ agree on x. Note that $x \in Imp$ because $x \in Imp_{i+1}$ and there is no assignment statement to x in the initial subsequence $T_1, T_2, \cdots, T_i$. Since $\hat{\sigma}_1$ and $\hat{\sigma}_1'$ agree on Imp, $\hat{\sigma}_1$ and $\hat{\sigma}_1'$ agree on x. Thus, $\hat{\sigma}_{i+1}$ and $\hat{\sigma}_1'$ agree on x. Since $\hat{\sigma}_{i+1}$ and $\hat{\sigma}_{\pi(i+1)}'$ do not agree on x, $\hat{\sigma}_1'$ and $\hat{\sigma}_{\pi(i+1)}'$ do not agree on x. Thus the execution of the initial subsequence $U_1, U_2, \cdots, U_{\pi(i+1)-1}$ executed an assignment statement to x and reached the beginning of $U_{\pi(i+1)}$. Let the assignment statement a be in U_k. Since x is an imported variable of $U_{\pi(i+1)}$, $U_{\pi(i+1)}$ has an incoming loop-independent flow edge or an incoming def-order edge $a \longrightarrow \ldots$, whose source is in U_k. Since T and U have isomorphic program dependence graphs, there is a corresponding loop-independent flow edge or a corresponding def-order edge from a vertex in $T_{\pi^{-1}(k)}$ to a vertex in T_{i+1}. Therefore, $\pi^{-1}(k) < i+1$. Because U_k has an assignment statement, a, to x, $T_{\pi^{-1}(k)}$ has a corresponding assignment statement to x, which contradicts a previous assumption that there is no assignment statement to x in the initial subsequence $T_1, T_2, \cdots, T_i$.

(2) Suppose there are assignment statements that assign to x in the initial subsequence $T_1, T_2, \cdots, T_i$. Let m be the largest number in $1, 2, \cdots, i$ such that T_m contains an assignment statement to x. Because x is an imported variable of T_{i+1}, T_{i+1} has an incoming loop-independent flow edge or an incoming def-order edge whose source is in T_m. Since T and U have isomorphic program dependence graphs, there is a corresponding loop-independent flow edge or a corresponding def-order edge from a vertex in $U_{\pi(m)}$ to a vertex in $U_{\pi(i+1)}$ and hence $\pi(m) < \pi(i+1)$. Note that $\hat{\sigma}_{m+1}$ and $\hat{\sigma}_{i+1}$ agree on x because there is no assignment statement to x in the subsequence $T_{m+1}, T_{m+2}, \cdots, T_i$. Note also that $x \in Exp_m$ because T_m has an outgoing loop-independent flow edge whose source is an assignment statement to x.

Since $m \le i$, by the induction hypothesis, $\hat{\sigma}_m$ and $\hat{\sigma}_{\pi(m)}'$ agree on Imp_m. Because $\hat{\sigma}_m$ and $\hat{\sigma}_{\pi(m)}'$ agree on Imp_m and T_m and $U_{\pi(m)}$ are corresponding subtrees, by the Equivalence Lemma, the execution states after executing T_m and $U_{\pi(m)}$, $\hat{\sigma}_{m+1}$ and $\hat{\sigma}_{\pi(m)+1}'$, agree on Exp_m, and hence they agree on x. Thus, $\hat{\sigma}_{i+1}$ and $\hat{\sigma}_{\pi(m)+1}'$ agree on x.

Because $\hat{\sigma}_{i+1}$ and $\hat{\sigma}_{\pi(i+1)}{}'$ do not agree on x, $\hat{\sigma}_{\pi(m)+1}{}'$ and $\hat{\sigma}_{\pi(i+1)}{}'$ do not agree on x. Thus the execution of the subsequence $U_{\pi(m)+1}, U_{\pi(m)+2}, \cdots, U_{\pi(i+1)-1}$ executed an assignment statement to x and reached the beginning of $U_{\pi(i+1)}$. Let the assignment statement a be in U_k, where $\pi(m) < k < \pi(i+1)$. Since x is an imported variable of $U_{\pi(i+1)}$, $U_{\pi(i+1)}$ has an incoming loop-independent flow edge or an incoming def-order edge $a \longrightarrow \ldots$, whose source is in U_k.

Since T and U have isomorphic program dependence graphs, there is a corresponding loop-independent flow edge or a corresponding def-order edge from a vertex in $T_{\pi^{-1}(k)}$ to a vertex in T_{i+1}. Therefore, $\pi^{-1}(k) < i+1$. Note that $\pi(m) < k < \pi(i+1)$ and both assignment statements to x in $U_{\pi(m)}$ and U_k can reach $U_{\pi(i+1)}$. Hence U_k has an incoming def-order edge whose source is in $U_{\pi(m)}$. Since T and U have isomorphic program dependence graphs, $T_{\pi^{-1}(k)}$ has a corresponding def-order edge whose source is in T_m. Therefore, $m < \pi^{-1}(k)$. Thus, $m < \pi^{-1}(k) < i+1$. Because there is an assignment statement to x in U_k and U_k and $T_{\pi^{-1}(k)}$ are corresponding subtrees, there is an assignment statement to x in $T_{\pi^{-1}(k)}$, which contradicts a previous assumption that m is the largest number in $1, 2, \cdots, i$ such that T_m contains an assignment statement to x.

We conclude that $\hat{\sigma}_{i+1}$ and $\hat{\sigma}_{\pi(i+1)}{}'$ agree on Imp_{i+1}. By the induction hypothesis of the structural induction, T_{i+1} and $U_{\pi(i+1)}$ compute the same sequence of values at each corresponding program point.

This completes the induction, so we conclude that T and U compute the same sequence of values at each corresponding program point.

Case 5. The operator at the root of T is the Program operator. Because $Imp = Imp_{stmt\_list}$, the proposition that T and U compute the same sequence of values at each corresponding program point follows directly from the induction hypothesis. □

4.2.2. A Strong Form of the Equivalence Theorem

The Strong Form of the Equivalence Theorem follows as a corollary of the Subtree Equivalence Lemma; it is simply the Subtree Equivalence Lemma specialized to the case when subtree T is the entire program P.

THEOREM. (STRONG FORM OF THE EQUIVALENCE THEOREM). *Suppose that P and Q are programs for which $G_P \approx G_Q$. If σ is a state on which P halts, then for any state σ' that agrees with σ on all variables for which there are initial-definition vertices in G_P: (1) Q halts on σ', (2) P and Q compute the same sequence of values at each corresponding program points, and (3) the final states agree on all variables for which there are final-use vertices in G_P.*

PROOF. Immediate from the Subtree Equivalence Lemma. □

Note that the differences among programs with isomorphic program dependence graphs are the order in which the statements occur in the program. The strong form of the Equivalence Theorem tells us that we may choose any one among those programs with isomorphic program dependence graphs as the "representative" of them. Therefore, in proving other theorems, such as the Slicing Theorem, we may assume the statements of the program appear in some particular order as needed to make the proof more tractable.

4.3. The Subtree Slicing Lemma

The Subtree Slicing Lemma characterizes the relationship between a subtree and a slice of the subtree in terms of the slice's imported and exported variables. The Lemma asserts that, for certain initial states, corresponding subtrees of a program and a slice of the program compute the same sequence of values at common program points.

LEMMA. (SUBTREE SLICING LEMMA). *Let Q be a slice of program P with respect to a set of vertices. Let T be a subtree of program P and U be the corresponding subtree of Q. If σ is a state on which T halts, then (1) U halts on σ' where σ and σ' agree on U's imported variables (as defined in the context given by Q), (2) T and U compute the same sequence of values at each program point of U, and (3) the final states agree on U's exported variables (as defined in the context given by Q).*

PROOF. By the Strong Form of the Equivalence Theorem, all programs with isomorphic PDGs compute the same sequence of values at each corresponding program point. We choose Q to be the version of the slice whose statements are in the same order as in P.

The proof is by structural induction on the abstract syntax of the programming language. The proof splits into five cases based on the abstract-syntax operator that appears at the root of T.

Throughout the proof, we use σ_1 and σ_1' to denote states that agree on U's imported variables, Imp_U. We use σ_i to denote a sequence of states in the execution of T initiated on σ_1, and we use σ_i' to denote the corresponding sequence of states in the execution of U initiated on σ_1'.

Case 1. The operator at the root of T is the Assign operator. Because T is a single assignment statement, either U is the tree *Null* or $U = T$. If U is *Null*, then $Imp_U = Exp_U = \emptyset$. Hence U always halts and the final states agree on Exp_U (since Exp_U is empty).

Now suppose $U = T$ and that U assigns to variable x as a function of variables $\{y_j\}$. The set Imp_U is either $\{y_j\}$ or $\{y_j\} \cup \{x\}$. (Imp_U is $\{y_j\} \cup \{x\}$ when U is the target of a def-order edge.) Since the value of the *exp* is a function of $\{y_j\}$; and $\{y_j\} \subseteq Imp_U$, evaluating *exp* in both σ_1 and σ_1' yields the same value because they agree on Imp_U. Exp_U is either $\emptyset$ or $\{x\}$. For any combination of these possibilities, σ_2 and σ_2' agree on x, and hence they agree on Exp_U.

Case 2. The operator at the root of T is the While operator. If the vertex corresponding to T's *exp* component is not in U, then U is the tree *Null*. If U is *Null*, then $Imp_U = Exp_U = \emptyset$. Hence U always halts and the final states agree on Exp_U.

We use Imp_{exp} and Exp_{exp} to denote the imported and exported variables of U's *exp* component, respectively; $Imp_{U_{stmt\_list}}$ and $Exp_{U_{stmt\_list}}$ denote the imported and exported variables of U's *stmt_list* component, respectively. We use σ_i and σ_i' to denote the execution states before executing the i^{th} iterations of the loops of T and U starting from two states that agree on Imp_U, σ_1 and σ_1', respectively.

Suppose the vertex corresponding to T's *exp* component is in U. Since T halts we may assume the execution of T halts after the j^{th} iteration, for some j. It is sufficient to show that (1) U also halts after the j^{th} iteration, (2) in each iteration, T and U compute the same sequence of values at each program point of U, and (3) the final states, σ_{j+1} and σ_{j+1}' agree on Exp_U. Because for a loop $Exp_U \subseteq Imp_U$,[5] it suffices to show that if σ_i and σ_i' agree on Imp_U then either T and U halt in the states σ_i and σ_i', respectively, or the i^{th} iterations compute the same sequence of values at each program point of U and result in the states σ_{i+1} and σ_{i+1}' that agree on Imp_U.

First, we show that $Imp_U = Imp_{exp} \cup Imp_{U_{stmt\_list}}$. It is clear that we could have written this with $\subseteq$, noting that $Imp_{U_{stmt\_list}}$ can include a variable x that is used at the target t of a loop-carried flow dependence edge where the dependence is carried by U. However, there then has to exist an incoming loop-independent

---

[5]If $x \in Exp_U$, then U contains an assignment a to x with an outgoing flow edge $a \longrightarrow_f b$. Because the loop may execute zero times, the assignment to x must be the target of a def-order edge $\ldots \longrightarrow_{do(b)} a$, hence $x \in Imp_U$.

flow edge to t, which implies that $v \in Imp_U$.

Let σ_i and σ_i' be states that agree on Imp_U. Therefore they agree on Imp_{exp}. Evaluating the condition (the *exp* component of U) in σ_i and σ_i' yields the same value. Hence, T and U compute the same (sequence of) values at the control predicate of the loop in the i^{th} iteration. If the condition evaluates to false, then both executions terminate in the states σ_i and σ_i', which agree on Exp_U.

Now suppose the condition evaluates to true. Let σ_i and σ_i' be states that agree on Imp_U; therefore they agree on $Imp_{U_{stmt\_list}}$. Now T_{stmt_list} and U_{stmt_list} are corresponding subtrees. Since T halts on σ_i, T_{stmt_list} also halts on σ_i. By the induction hypothesis, (1) U_{stmt_list} halts on σ_i', (2) during the i^{th} iteration T_{stmt_list} and U_{stmt_list} compute the same sequence of values at each program point of U_{stmt_list}, and (3) the final states, σ_{i+1} and σ_{i+1}', agree on $Exp_{U_{stmt\_list}}$. If σ_{i+1} and σ_{i+1}' do not also agree on Imp_U, then let $x \in Imp_U$ be a variable on which they disagree (so $x \notin Exp_{U_{stmt\_list}}$). Now, by assumption, σ_i and σ_i' agree on Imp_U; therefore, at least one of the two executions of T_{stmt_list} and U_{stmt_list}, respectively, executed an assignment statement a that assigned a value to x and reached the end of the *stmt_list*. There are two cases to consider:

(1) One possibility is that $x \in Imp_U$ because x is used in a condition or statement b that is the target of an incoming flow edge *If this were the case, then there must be a loop-carried flow edge* $a \longrightarrow_{lc(T)} b$ *or* $a \longrightarrow_{lc(U)} b$*, depending on whether* T_{stmt_list} *or* U_{stmt_list} *executed* a*. However, in either case,* a *is in* U *because* b *is in* U*; therefore,* a *is in* U_{stmt_list} *and* $x \in Exp_{U_{stmt\_list}}$*, which contradicts our previous assumption.*

(2) The other possibility is that $x \in Imp_U$ because the U_{stmt_list} has an incoming def-order edge $\ldots \longrightarrow_{do(c)} d$. However, this implies that there is an outgoing flow edge $a \longrightarrow_f c$ from T_{stmt_list} or U_{stmt_list}, depending on whether T_{stmt_list} or U_{stmt_list} executed a. In either case, however, a must be in U because c is in U; therefore, a is in U_{stmt_list} and $x \in Exp_{U_{stmt\_list}}$, which contradicts our previous assumption.

We conclude that σ_{i+1} and σ_{i+1}' agree on Imp_U. Therefore U halts after the j^{th} iteration, during each iteration T and U compute the same sequence of values at each program point of U, and σ_{j+1} and σ_{j+1}' agree on Exp_U.

Case 3. The operator at the root of T is the IfThenElse operator. If the vertex corresponding to T's *exp* component is not in U, then U is the tree *Null*. If U is *Null*, $Imp_U = Exp_U = \emptyset$. Therefore, U always halts and the final states agree on Exp_U.

Suppose the vertex corresponding to T's *exp* component is in U. Evaluating the condition (the *exp* component of U) in σ_1 and σ_1' yields the same value. Therefore, T and U compute the same (sequence of) values at the control predicate of the IfThenElse statement.

Without loss of generality, assume that the condition evaluates to true. We use T_{true}, T_{false}, U_{true}, and U_{false} to denote the respective branches of T and U.

When execution is initiated in state σ_1, T terminates in σ_2; consequently T_{true} also terminates in σ_2. Since σ_1 and σ_1' agree on Imp_U and $Imp_{U_{true}} \subseteq Imp_U$, σ_1 and σ_1' agree on $Imp_{U_{true}}$. Because T_{true} and U_{true} are corresponding subtrees, the induction hypothesis tells us that, when execution is initiated in state σ_1', (1) U_{true} terminates in state σ_2' (hence U terminates in σ_2'), (2) T_{true} and U_{true} compute the same sequence of values at each program point of U_{true} (hence T and U compute the same sequence of values at each program point of U), and (3) σ_2 and σ_2' agree on $Exp_{U_{true}}$.

Note that $Exp_U = Exp_{U_{true}} \cup Exp_{U_{false}}$ so what remains to be shown is that σ_2 and σ_2' agree on $Exp_{U_{false}}$. If σ_2 and σ_2' do not also agree on $Exp_{U_{false}}$, then let $x \in Exp_{U_{false}}$ be a variable on which they disagree (so

$x \notin Exp_{U_{true}}$). Because $x \in Exp_{U_{false}}$, there is an assignment statement a in the false branch of U that assigns to x and is the source of an outgoing flow edge from that branch (say $a \longrightarrow_f b$).

We must consider whether it is possible that $x \notin Imp_U$. By assumption, $x \notin Exp_{U_{true}}$. Consider an execution path p, from the beginning of the program to the beginning of statement U, that does not include the back-edges of any loops. Let c be the last assignment statement that assigns to x along p, or, if no such statement exists, let c be the initial-definition vertex for x. Because we can extend path p to first follow the true branch of U and then continue from the join point of U via the path by which a reaches b, we deduce that there is a dependence $c \longrightarrow_f b$. By construction, vertex c occurs to the left of a, hence $c \longrightarrow_{do(b)} a$. We conclude that $x \in Imp_U$.

Since $x \in Imp_U$, σ_1 and σ_1' agree on x. Because σ_2 and σ_2' disagree on x, at least one of the two executions of the true branches of T and U, respectively, executed an assignment statement d that assigned a value to x and reached the end of the true branch. But this implies the existence of a flow edge $d \longrightarrow_f b$ in either T or U, depending on whether T_{true} or U_{true} executed the assignment to x. In either case, the flow edge $d \longrightarrow_f b$ is in Q since b is in Q, and hence d is in U_{true}. Therefore, $x \in Exp_{U_{true}}$, which contradicts a previous assumption. We conclude that σ_2 and σ_2' agree on $Exp_{U_{false}}$. This, together with the fact that σ_2 and σ_2' agree on $Exp_{U_{true}}$, means that they agree on Exp_U.

Case 4. The operator at the root of T is the StmtList operator. Let $T_1, T_2, \cdots, T_n$ denote the immediate subtrees of T and $U_1, U_2, \cdots, U_n$ denote the corresponding subtrees of U. (Note that some of the U_i may be the tree *Null*.) We use σ_i and σ_i' to denote the execution states before executing T_i and U_i, respectively; we use Imp_{U_i} and Exp_{U_i} to denote the imported and exported variables, respectively, of U_i; and we use $Imp_{U_{1..i}}$ and $Exp_{U_{1..i}}$ to denote the imported and exported variables, respectively, of the initial subsequence $U_1, U_2, \cdots, U_i$. (Although the imported and exported variables for subsequences were not part of the definition in Section 4.1, we intend the obvious extension: the imported variables of a subsequence are defined in terms of incoming edges whose targets are inside the subsequence; the exported variables of a subsequence are defined in terms of outgoing edges whose sources are inside the subsequence).

The proof of this case is by induction over the initial subsequences of U. We want to show that for all i, $1 \le i \le n$, if σ_1 is a state on which T halts and σ_1 and σ_1' agree on $Imp_{U_{1..i}}$, then $T_{1..i}$ and $U_{1..i}$ terminate in σ_{i+1} and σ_{i+1}', respectively, and $T_{1..i}$ and $U_{1..i}$ compute the same sequence of values at each program point of $U_{1..i}$ and σ_{i+1} and σ_{i+1}' agree on $Exp_{U_{1..i}}$.

Base case. $n = 1$. The proposition follows immediately from the induction hypothesis of the structural induction.

Induction step. The induction hypothesis is: if σ_1 and σ_1' agree on $Imp_{U_{1..i}}$ and T halts on σ_1, then (1) $T_{1..i}$ and $U_{1..i}$ terminate in σ_{i+1} and σ_{i+1}', respectively, (2) $T_{1..i}$ and $U_{1..i}$ compute the same sequence of values at each program point of $U_{1..i}$, and (3) σ_{i+1} and σ_{i+1}' agree on $Exp_{U_{1..i}}$. Thus, if $\hat{\sigma}_1$ and $\hat{\sigma}_1'$ are arbitrary states that agree on $Imp_{U_{1..i+1}}$ and T halts on $\hat{\sigma}_1$, we need to show that $T_{1..i+1}$ and $U_{1..i+1}$ terminate in $\hat{\sigma}_{i+2}$ and $\hat{\sigma}_{i+2}'$, respectively, and $T_{1..i+1}$ and $U_{1..i+1}$ compute the same sequence of values at each program point of $U_{1..i+1}$ and $\hat{\sigma}_{i+2}$ and $\hat{\sigma}_{i+2}'$ agree on $Exp_{U_{1..i+1}}$.

Note that $Imp_{U_{1..i}} \subseteq Imp_{U_{1..i+1}}$, which means that $\hat{\sigma}_1$ and $\hat{\sigma}_1'$ agree on $Imp_{U_{1..i}}$, and thus, by the induction hypothesis, (1) $T_{1..i}$ and $U_{1..i}$ terminate in $\hat{\sigma}_{i+1}$ and $\hat{\sigma}_{i+1}'$, respectively, (2) $T_{1..i}$ and $U_{1..i}$ compute the same sequence of values at each program point of $U_{1..i}$, and (3) $\hat{\sigma}_{i+1}$ and $\hat{\sigma}_{i+1}'$ agree on $Exp_{U_{1..i}}$.

First, we must show that $\hat{\sigma}_{i+1}$ and $\hat{\sigma}_{i+1}'$ agree on $Imp_{U_{i+1}}$. Any variable $x \in Imp_{U_{i+1}}$ on which $\hat{\sigma}_{i+1}$ and $\hat{\sigma}_{i+1}'$ disagree must be in $Imp_{U_{1..i+1}}$ (if not, x would be in $Exp_{U_{1..i}}$ on which $\hat{\sigma}_{i+1}$ and $\hat{\sigma}_{i+1}'$ agree). By assumption, $\hat{\sigma}_1$ and $\hat{\sigma}_1{}'$ agree on $Imp_{U_{1..i+1}}$; consequently, at least one of the two executions of $T_{1..i}$ and $U_{1..i}$ performed an assignment statement, a, that assigned to x and reached the end of T_i or U_i, depending on whether $T_{1..i}$ or $U_{1..i}$ performed the assignment. There are now two cases to consider:

(1) One possibility is that $x \in Imp_{U_{i+1}}$ because x is used in a condition or statement b that is the target of one of U_{i+1}'s incoming flow edges. In this case, there is a flow edge: $a \longrightarrow_f b$ in T or U, depending on whether T or U performed the assignment. In either case, a is in U because b is in U. Therefore, this edge must be in U. This implies that $x \in Exp_{U_{1..i}}$, so $\hat{\sigma}_{i+1}$ and $\hat{\sigma}_{i+1}{}'$ must agree on x, which contradicts our assumption that they disagree on x.

(2) The other possibility is that $x \in Imp_{U_{i+1}}$ because there is an incoming def-order edge, $\ldots \longrightarrow_{do(d)} c$, to U_{i+1}. However, this implies that there is an outgoing flow edge of : $a \longrightarrow_f d$ in T or U depending whether T or U performed the assignment, a. In either case, a is in U because d is in Q. Therefore, this flow edge $a \longrightarrow_f d$ is in Q. As in the previous case, this implies that $x \in Exp_{U_{1..i}}$, so $\hat{\sigma}_{i+1}$ and $\hat{\sigma}_{i+1}{}'$ must agree on x, which contradicts our assumption that they disagree on x.

We conclude that $\hat{\sigma}_{i+1}$ and $\hat{\sigma}_{i+1}{}'$ agree on $Imp_{U_{i+1}}$.

Because T terminates on $\hat{\sigma}_1$, T_{i+1} must terminate on $\hat{\sigma}_{i+1}$. Because $\hat{\sigma}_{i+1}$ and $\hat{\sigma}_{i+1}{}'$ agree on $Imp_{U_{i+1}}$, the induction hypothesis of the structural induction tells us (1) the execution of U_{i+1} on $\hat{\sigma}_{i+1}{}'$ halts, (2) T_{i+1} and U_{i+1} compute the same sequence of values at each program point of U_{i+1} and $\hat{\sigma}_{i+2}$, and (3) $\hat{\sigma}_{i+2}{}'$ agree on $Exp_{U_{i+1}}$.

The final step is to show that $\hat{\sigma}_{i+2}$ and $\hat{\sigma}_{i+2}{}'$ agree on $Exp_{U_{1..i+1}}$. Note that $Exp_{U_{1..i+1}} \subseteq Exp_{U_{1..i}} \cup Exp_{U_{i+1}}$. Now suppose there is a variable $x \in Exp_{U_{1..i+1}}$ on which $\hat{\sigma}_{i+2}$ and $\hat{\sigma}_{i+2}{}'$ disagree (in particular, $x \notin Exp_{U_{i+1}}$). Therefore, $x \in Exp_{U_{1..i}}$. By the induction hypothesis, $\hat{\sigma}_{i+1}$ and $\hat{\sigma}_{i+1}{}'$ agree on $Exp_{U_{1..i}}$, so at least one of the two executions of T_{i+1} and U_{i+1} performed an assignment statement, a, that assigned to x and reached the end of T_{i+1} or U_{i+1}, depending on whether T_{i+1} or U_{i+1} performed the assignment. If T_{i+1} performed the assignment, since $x \in Exp_{U_{1..i+1}} \subseteq Exp_{T_{1..i+1}}$, there must also be an outgoing flow edge $a \longrightarrow_f \ldots$ from T_{i+1}. Since $U_{1..i+1}$ is a slice of $T_{1..i+1}$ and $x \in Exp_{U_{1..i+1}}$, therefore the flow edge $a \longrightarrow_f \ldots$ is in $U_{1..i+1}$. This implies that $x \in Exp_{U_{i+1}}$, so $\hat{\sigma}_{i+2}$ and $\hat{\sigma}_{i+2}{}'$ must agree on x, which contradicts our assumption that they disagree on x. If U_{i+1} performed the assignment, since $x \in Exp_{U_{1..i+1}}$, there must also be an outgoing flow edge $a \longrightarrow_f \ldots$ from U_{i+1}. This implies that $x \in Exp_{U_{i+1}}$, so $\hat{\sigma}_{i+2}$ and $\hat{\sigma}_{i+2}{}'$ must agree on x, which contradicts our assumption that they disagree on x.

This completes the induction, so we conclude that U terminates on $\sigma_1{}'$, T and U compute the same sequence of values at each program point of U, and σ_{n+1} and $\sigma_{n+1}{}'$ agree on Exp_U.

Case 5. The operator at the root of T is the Program operator. Because $Imp_U = Imp_{U_{stmt\_list}}$ and $Exp_U = Exp_{U_{stmt\_list}}$, we conclude from the induction hypothesis that U terminates on σ', T and U compute the same sequence of values at each program point of U, and σ_2 and $\sigma_2{}'$ agree on Exp_U. □

4.4. The Slicing Theorem

The Slicing Theorem follows as a corollary of the Subtree Slicing Lemma; it is simply the Subtree Slicing Lemma specialized to the case when subtree T is the entire program P.

THEOREM. (SLICING THEOREM). *Let Q be a slice of program P with respect to a set of vertices. If σ is a state on which P halts, then for any state σ' that agrees with σ on all variables for which there are initial-definition vertices in G_Q: (1) Q halts on σ', (2) P and Q compute the same sequence of values at each* *program point of Q, and (3) the final states agree on all variables for which there are final-use vertices in* G_Q.

PROOF. Immediate from the Subtree Slicing Lemma. □

5. THE TERMINATION THEOREM

The Slicing Theorem tells us that if a program terminates on some initial state then (on sufficiently similar initial states) the program's slices also terminate. The Termination Theorem looks at this relationship from the opposite point of view; it tells us that if a program is decomposed into two slices, the termination of the slices on some states implies the termination of the program on a similar state. (It is straightforward to generalize the theorem to the case where the program is decomposed into more than two slices.)

5.1. The Subtree Termination Lemma

As in the Slicing Theorem, the proof of the Termination Theorem relies on a lemma about subtrees. The Subtree Termination Lemma states that if a program is decomposed into two slices, a subtree of the program will terminate on a state when the corresponding subtrees of the two slices terminate on some similar states.

LEMMA. (SUBTREE TERMINATION LEMMA). *Let P be a program. Suppose X and Y are sets of vertices such that* $G_P = G_P / X \cup G_P / Y$. *Let T be a subtree of program P and U and V be the corresponding subtrees of P / X and P / Y, respectively. Suppose* σ_U *is a state on which U halts, and* σ_V *is a state on which V halts. Then for any state* σ*, where* σ *and* σ_U *agree on U's imported variables and* σ *and* σ_V *agree on V's imported variables, T halts on* σ.

PROOF. By the Equivalence Theorem, all programs with isomorphic program dependence graphs are strongly equivalent. We choose P / X and P / Y to be the versions of the slices whose statements are in the same order as in P.

The proof is by structural induction on the abstract syntax of the programming language. The proof splits into five cases based on the abstract-syntax operator that appears at the root of T.

Case 1. The operator at the root of T is the Assign operator. Since $G_P = G_P / X \cup G_P / Y$, $T = U$ or $T = V$. Without loss of generality, suppose $T = U$. Because σ and σ_U agree on U's imported variables and U halts on σ_U, by the Self-Equivalence Lemma T halts on σ.

Case 2. The operator at the root of T is the While operator. Since $G_P = G_P / X \cup G_P / Y$, if U is a *Null* tree, then $T = V$. Similarly, if V is a *Null* tree, then $T = U$. Without loss of generality, suppose $T = U$. Because σ and σ_U agree on U's imported variables and U halts on σ_U, by the Self-Equivalence Lemma T halts on σ.

Now suppose both U and V are not *Null* trees. Since U halts on σ_U, we may assume that the execution of U halts after the j^{th} iteration, for some j. We prove that T and V halt on σ and σ_V, respectively, after exactly j iterations. Because for a loop $Exp_U \subseteq Imp_U$, it suffices to show that if σ and σ_U agree on U's imported variables and σ and σ_V agree on V's imported variables, then either T, U, and V halt in the states σ, σ_U, and σ_V, respectively, or T, U, and V successfully finish one iteration and the execution states that result after one iteration of the loops (σ', σ_U', and σ_V', respectively) are ones such that σ' and σ_U' agree on U's imported variables and σ' and σ_V' agree on V's imported variables.

We use T_{stmt_list}, U_{stmt_list}, and V_{stmt_list} to denote the *stmt_list* components of T, U, and V, respectively. Note that T_{stmt_list}, U_{stmt_list}, and V_{stmt_list} are corresponding subtrees of P, P / X, and P / Y, respectively.

Because σ and σ_U agree on U's imported variables, evaluating the control predicates in σ and σ_U yields the same value. Because σ and σ_V agree on V's imported variables, evaluating the control predicates in σ and σ_V yields the same value. If the control predicate evaluates to false, then T, U, and V halt in the states σ, σ_U, and σ_V, respectively.

Now suppose the control predicate evaluates to true. Because σ and σ_U agree on U's imported variables and the imported variables of U_{stmt_list} are a subset of U's imported variables, σ and σ_U agree on the imported variables of U_{stmt_list}. Similarly, σ and σ_V agree on the imported variables of V_{stmt_list}. Note that T_{stmt_list}, U_{stmt_list}, and V_{stmt_list} are corresponding subtrees of P, P/X, and P/Y, respectively. Because U_{stmt_list} and V_{stmt_list} halt on σ_U and σ_V, respectively, by the induction hypothesis, T_{stmt_list} halts on σ. Therefore, T, U, and V successfully finish one iteration. Let σ', σ_U', and σ_V' denote the execution states of T, U, and V after one iteration of the loop, respectively. By the Subtree Slicing Lemma, σ' and σ_U' agree on U's exported variables and σ' and σ_V' agree on V's exported variables. By the same argument as in the proof of the Subtree Slicing Lemma, Case 2, σ' and σ_U' agree on U's imported variables. Similarly, σ' and σ_V' agree on V's imported variables. We conclude that T, U, and V halt on σ, σ_U, and σ_V, respectively, after the j^{th} iteration.

Case 3. The operator at the root of T is the IfThenElse operator. Since $G_P = G_P/X \cup G_P/Y$, if U is a *Null* tree, then $T = V$. Similarly, if V is a *Null* tree, then $T = U$. Without loss of generality, suppose $T = U$. Because σ and σ_U agree on U's imported variables and U halts on σ_U, by the Self-Equivalence Lemma T halts on σ.

Now suppose both U and V are not *Null* trees. Because σ and σ_U agree on U's imported variables, evaluating the control predicates in σ and σ_U yields the same value. Because σ and σ_V agree on V's imported variables, evaluating the control predicates in σ and σ_V yields the same value. Without loss of generality, we may assume the control predicate evaluates to true.

We use T_{true}, T_{false}, U_{true}, U_{false}, V_{true}, and V_{false} to denote the respective branches of T, U, and V, respectively. Note that T_{true}, U_{true}, and V_{true} are corresponding subtrees of P, P/X, and P/Y, as are T_{false}, U_{false}, and V_{false}.

When execution is initiated in state σ_U, U terminates; consequently, U_{true} also terminates. Similarly, when execution is initiated in state σ_V, V terminates; consequently, V_{true} also terminates. Because σ and σ_U agree on U's imported variables and the imported variables of U_{true} are a subset of U's imported variables, σ and σ_U agree on the imported variables of U_{true}. Similarly, σ and σ_V agree on the imported variables of V_{true}. Because T_{true}, U_{true}, and V_{true} are corresponding subtrees of P, P/X, and P/Y, respectively, the induction hypothesis tells us that the T_{true} halts on σ. Hence, T halts on σ.

Case 4. The operator at the root of T is the StmtList operator. Let $T_1, T_2, \cdots, T_n$ denote the immediate subtrees of T, $U_1, U_2, \cdots, U_n$ denote the corresponding subtrees of U, and $V_1, V_2, \cdots, V_n$ denote the corresponding subtrees of V. Let $T_{1..i}$, $U_{1..i}$, and $V_{1..i}$ denote the initial subsequences $T_1, T_2, \cdots, T_i$, $U_1, U_2, \cdots, U_i$, and $V_1, V_2, \cdots, V_i$, respectively.

The proof of this case is by induction over the initial subsequences of T. We want to show that for all i, $1 \le i \le n$, if $U_{1..i}$ and $V_{1..i}$ halt on σ_U and σ_V, respectively, then $T_{1..i}$ halts on σ where σ and σ_U agree on the imported variables of $U_{1..i}$, and σ and σ_V agree on the imported variables of $V_{1..i}$.

Base case. $n = 1$. The proposition follows immediately from the induction hypothesis of the structural induction.

Induction step. The induction hypothesis is: if $U_{1..i}$ and $V_{1..i}$ halt on σ_U and σ_V, respectively, then $T_{1..i}$ halts on σ where σ and σ_U agree on the imported variables of $U_{1..i}$, and σ and σ_V agree on the imported variables of $V_{1..i}$. Thus, if $\hat{\sigma}_U$ and $\hat{\sigma}_V$ are arbitrary states on which $U_{1..i+1}$ and $V_{1..i+1}$ halt, respectively, we need to show that $T_{1..i+1}$ halts on $\hat{\sigma}$ where $\hat{\sigma}$ and $\hat{\sigma}_U$ agree on the imported variables of $U_{1..i+1}$, and $\hat{\sigma}$ and $\hat{\sigma}_V$ agree on the imported variables of $V_{1..i+1}$.

Note that the imported variables of $U_{1..i}$ are a subset of the imported variables of $U_{1..i+1}$, which means that $\hat{\sigma}$ and $\hat{\sigma}_U$ agree on the imported variables of $U_{1..i}$. Similarly, $\hat{\sigma}$ and $\hat{\sigma}_V$ agree on the imported variables of $V_{1..i}$. Thus, by the induction hypothesis, $T_{1..i}$ halts on $\hat{\sigma}$. Let $\hat{\sigma}'$, $\hat{\sigma}_U'$, and $\hat{\sigma}_V'$ be the execution states after executing the initial subsequences $T_{1..i}$, $U_{1..i}$, and $V_{1..i}$, respectively. By the Subtree Slicing Lemma, $\hat{\sigma}'$ and $\hat{\sigma}_U'$ agree on the exported variables of $U_{1..i}$. By the same argument as in the proof of the Subtree Slicing Lemma, Case 4, $\hat{\sigma}'$ and $\hat{\sigma}_U'$ agree on the imported variables of U_{i+1}. Similarly, $\hat{\sigma}'$ and $\hat{\sigma}_V'$ agree on the imported variables of V_{i+1}. Note that T_{i+1}, U_{i+1}, and V_{i+1} are corresponding subtrees of P, P/X, and P/Y, respectively. Because U_{i+1} and V_{i+1} halt on $\hat{\sigma}_U'$ and $\hat{\sigma}_V'$, respectively, by the induction hypothesis of the structural induction, T_{i+1} halts on $\hat{\sigma}'$. Now we have proved that $T_{1..i}$ halts on $\hat{\sigma}$, resulting in $\hat{\sigma}'$, and T_{i+1} halts on $\hat{\sigma}'$. Therefore, $T_{1..i+1}$ halts on $\hat{\sigma}$.

This completes the induction, so we conclude that T halts on σ.

Case 5. The operator at the root of T is the Program operator. Because the imported variables of U are the same as the imported variables of U_{stmt_list} and the imported variables of V are the same as the imported variables of V_{stmt_list}, by the induction hypothesis of the structural induction, T halts on σ. □

5.2. The Termination Theorem

The Termination Theorem follows as a corollary of the Subtree Termination Lemma; it is simply the Subtree Termination Lemma specialized to the case when subtree T is the entire program P.

THEOREM. (TERMINATION THEOREM). *Let P be a program. Suppose X and Y are sets of vertices such that $G_P = G_P/X \cup G_P/Y$. If P/X and P/Y halt on a state σ, then P halts on σ as well.*

PROOF. Immediate from the Subtree Termination Lemma. □

Note that the Termination Theorem and clause (1) of Slicing Theorem are complementary: clause (1) of the Slicing Theorem asserts that if a program terminates then each slice also terminates; the Termination Theorem asserts that when a program can be decomposed into two slices, if each slice terminates then the program terminates. We can then apply clause (2) of the Slicing Theorem to conclude that the two slices (collectively) compute the same sequence of values as the entire program.

The following Corollary generalizes the Termination Theorem to the case when the program is decomposed into three slices. It is used in the proof of the Integration Theorem that is given in the next section; the integrated program that is the subject of the proof is formed by taking the union of three slices.

COROLLARY. *Let P be a program. Suppose X, Y, and Z are sets of vertices such that $G_P = G_P/X \cup G_P/Y \cup G_P/Z$. If P/X, P/Y, and P/Z halt on a state σ, then P halts on σ as well.*

PROOF. >From the Decomposition Lemma, we have $G_P/X \cup G_P/Y = G_P/(X \cup Y)$. Let $P/(X \cup Y)$ denote a program whose program dependence graph is (isomorphic to) $G_P/(X \cup Y)$. Since P/X and P/Y halt on σ, by the Subtree Termination Lemma, $P/(X \cup Y)$ halts on σ. Similarly, $G_P = G_P/X \cup G_P/Y \cup G_P/Z = G_P/(X \cup Y) \cup G_P/Z$. Since $P/(X \cup Y)$ and P/Z halt on σ, P halts on σ. □

6. THE SEMANTICS OF PROGRAM INTEGRATION

An algorithm for integrating several related, but different variants of a base program (or determining whether the variants incorporate interfering changes) has been presented in [512]. The algorithm presented there, called Integrate, takes as input three programs A, B, and *Base*, where A and B are two variants of *Base*. As we show below, whenever the changes made to *Base* to create A and B do not “interfere” (in the sense defined below), Integrate produces a program M that exhibits the changed execution behavior of A and B with respect to *Base* as well as the execution behavior preserved in all three versions.

The first step of Integrate determines slices that represent a safe approximation to the changed computation threads of A and B and the computation threads of *Base* preserved in both A and B; the second step combines these slices to form the merged graph G_M; the third step tests G_M for interference.

Step 1: Determining changed and preserved computation threads

If the slice of variant G_A at vertex v differs from the slice of G_{Base} at v, then G_A and G_{Base} may compute different values at v. In other words, vertex v is a site that potentially exhibits changed behavior in the two programs. Thus, we define the *affected points* of G_A with respect to G_{Base}, denoted by $AP_{A,\ Base}$, to be the subset of vertices of G_A whose slices in G_{Base} and G_A differ $AP_{A,\ Base} = \{ v \mid v \in V(G_A) \quad (G_{Base} / v \neq G_A / v) \}$. We define $AP_{B,\ Base}$ similarly. It follows that the slices $G_A / AP_{A,\ Base}$ and $G_B / AP_{B,\ Base}$ capture the respective computation threads of A and B that differ from *Base*.

The preserved computation threads of *Base* in A correspond to the slice $G_{Base} / \overline{AP}_{A,\ Base}$, where $\overline{AP}_{A,\ Base}$ is the complement of $AP_{A,\ Base}$: $\overline{AP}_{A,\ Base} = V(G_A) - AP_{A,\ Base}$. We define $\overline{AP}_{B,\ Base}$ similarly. Thus, the unchanged computation threads common to both A and B are captured by the following slice: $G_{Base} / (\overline{AP}_{A,\ Base} \cap \overline{AP}_{B,\ Base})$.

Step 2: Forming the merged graph

The merged program dependence graph, G_M, is formed by unioning the three slices that represent the changed and preserved computation threads of the two variants:

$$G_M = (G_A / AP_{A,\ Base}) \cup (G_B / AP_{B,\ Base}) \cup (G_{Base} / (\overline{AP}_{A,\ Base} \cap \overline{AP}_{B,\ Base})).$$

Step 3: Testing for interference

There are two possible ways by which the graph G_M may fail to represent a satisfactory integrated program; both types of failure are referred to as “interference.” The first interference criterion is based on a comparison of slices of G_A, G_B, and G_M. The slices $G_A / AP_{A,\ Base}$ and $G_B / AP_{B,\ Base}$ represent the changed computation threads of programs A and B with respect to *Base*. A and B interfere if G_M does not preserve these slices; that is, there is interference of this kind if either $G_M / AP_{A,\ Base} \neq G_A / AP_{A,\ Base}$ or $G_M / AP_{B,\ Base} \neq G_B / AP_{B,\ Base}$.

The final step of the integration method involves reconstituting a program from the merged program dependence graph. However, it is possible that there is no such program; that is, the merged graph may be an infeasible program dependence graph. This is the second kind of interference that may occur. (The reader is referred to [5] for a discussion of reconstructing a program from the merged program dependence graph and the inherent difficulties of this problem.)

If neither kind of interference occurs, one of the programs that corresponds to the graph G_M is returned as the result of the integration operation.

6.1. The Integration Theorem

The Slicing and Termination Theorems, together with the definition of the merged graph G_M can be used to prove a theorem that characterizes the execution behavior of the integrated program in terms of the behaviors of the base program and the two variants:

THEOREM. (INTEGRATION THEOREM). *If A and B are two variants of Base for which integration succeeds (and produces program M), then for any initial state σ on which A, B, and Base all halt, (1) M halts on σ, (2) if x is a variable defined in the final state of A for which the final states of A and Base disagree, then the final state of M agrees with the final state of A on x, (3) if y is a variable defined in the final state of B for which the final states of B and Base disagree, then the final state of M agrees with the final state of B on y, and (4) if z is a variable on which the final states of A, B, and Base agree, then the final state of M agrees with the final state of Base on z.*

Restated less formally, *M* preserves the changed behaviors of both *A* and *B* (with respect to *Base*) as well as the unchanged behavior of all three.

The merged program dependence graph, G_M, is formed by unioning the three slices $G_A / AP_{A,\,Base}$, $G_B / AP_{B,\,Base}$, and $G_{Base} / (\overline{AP}_{A,\,Base} \cap \overline{AP}_{B,\,Base})$. Because the premise of the theorem is that integration succeeds, we know that $G_M / AP_{A,\,Base} = G_A / AP_{A,\,Base}$ and $G_M / AP_{B,\,Base} = G_B / AP_{B,\,Base}$. One detail that must be shown is that, in testing G_M for interference, it is unnecessary to test whether $G_{Base} / (\overline{AP}_{A,\,Base} \cap \overline{AP}_{B,\,Base}) = G_M / (\overline{AP}_{A,\,Base} \cap \overline{AP}_{B,\,Base})$.

This matter is addressed by the Preserved Behavior Lemma, stated and proven below, which shows that, regardless of whether or not the integration algorithm detects interference, the slice $G_{Base} / (\overline{AP}_{A,\,Base} \cap \overline{AP}_{B,\,Base})$ is always preserved in G_M.

LEMMA. *If* $w \in AP_{A,\,Base}$, *then* $w \notin G_{Base} / \overline{AP}_{A,\,Base}$.

PROOF. >From the definition, $AP_{A,\,Base} = \{ v \in V(A) \mid (G_{Base} / v \neq G_A / v) \}$, so $\overline{AP}_{A,\,Base} = \{ v \in V(A) \mid (G_{Base} / v = G_A / v) \}$. Using the Decomposition Lemma, we have:

$$G_{Base} / \overline{AP}_{A,\,Base} = G_{Base} / \{ v \in V(A) \mid (G_{Base} / v = G_A / v) \}$$

$$= \bigcup_{v \in V(A) \mid (G_{Base} / v = G_A / v)} G_{Base} / v$$

But if for some v, $w \in V(G_{Base} / v)$, then $G_{Base} / w \subseteq G_{Base} / v$; because $G_{Base} / w \neq G_A / w$, $G_{Base} / v \neq G_A / v$. Hence $w \notin G_{Base} / \overline{AP}_{A,\,Base}$. □

LEMMA (PRESERVED BEHAVIOR LEMMA). *Let*
$G_M = (G_A / AP_{A,\,Base}) \cup (G_B / AP_{B,\,Base}) \cup (G_{Base} / (\overline{AP}_{A,\,Base} \cap \overline{AP}_{B,\,Base}))$. *Then*
$G_{Base} / (\overline{AP}_{A,\,Base} \cap \overline{AP}_{B,\,Base}) = G_M / (\overline{AP}_{A,\,Base} \cap \overline{AP}_{B,\,Base})$.

PROOF. Let $PRE = G_{Base} / (\overline{AP}_{A,\,Base} \cap \overline{AP}_{B,\,Base})$ and $PRE' = G_M / (\overline{AP}_{A,\,Base} \cap \overline{AP}_{B,\,Base})$. Suppose $PRE \neq PRE'$. Because G_M is created by unioning *PRE* with $G_A / AP_{A,\,Base}$ and $G_B / AP_{B,\,Base}$, and the slices that generate *PRE* and *PRE′* are both taken with respect to the same set, $\overline{AP}_{A,\,Base} \cap \overline{AP}_{B,\,Base}$, it must be that $PRE \subset PRE'$.

Thus, there are three cases to consider: either *PRE′* contains an additional vertex, an additional control or flow edge (in the latter case either loop independent or loop carried), or an additional def-order edge.

Case 1. *PRE′* contains an additional vertex. Because the slices that generate *PRE* and *PRE′* are both taken with respect to the set, $\overline{AP}_{A,\,Base} \cap \overline{AP}_{B,\,Base}$, *PRE′* can only contain an additional vertex *v* if there is an additional control or flow edge $v \longrightarrow_{c,\,f} w$ whose target *w* is an element of both *PRE* and *PRE′*. Thus, this case reduces to the one that follows, in which *PRE′* contains an additional flow edge.

Case 2. *PRE′* contains an additional control or flow edge. The slice operation is a backward closure; because the slices that generate *PRE* and *PRE′* are both taken with respect to the same set, namely $\overline{AP}_{A,\,Base} \cap \overline{AP}_{B,\,Base}$, if *PRE′* were to contain control or flow edges not in *PRE*, then there is at least one such edge whose target vertex occurs in both *PRE* and *PRE′*. That is, there is at least one edge $e \colon v \longrightarrow w$, where $e \in E(PRE')$, v, $w \in V(PRE')$, $w \in V(PRE)$, and $v \notin V(PRE)$.

Because G_M is created by a graph union, e must occur in $E(G_A / AP_{A,\,Base})$, $E(G_B / AP_{B,\,Base})$, or both. Without loss of generality, assume that $e \in E(G_A / AP_{A,\,Base})$, so that $e \in E(G_A)$.

The slice operation is a backward closure, so $e \notin E(PRE)$ and $w \in V(PRE)$ imply $e \notin E(G_{Base})$. Taking this together with the previous observation that $e \in E(G_A)$, we conclude, from the definition of $AP_{A,\,Base}$, that $w \in V(AP_{A,\,Base})$.

This yields a contradiction as follows. By the previous lemma, $w \in V(AP_{A,\,Base})$ implies that $w \notin V(G_{Base} / \overline{AP}_{A,\,Base})$. However, $G_{Base} / (\overline{AP}_{A,\,Base} \cap \overline{AP}_{B,\,Base}) \subseteq G_{Base} / \overline{AP}_{A,\,Base}$, which means that $w \notin V(PRE)$.

Case 3. PRE' contains an additional def-order edge. Suppose $E(PRE')$ contains a def-order edge $e\colon v \longrightarrow_{do(u)} w$ that does not occur in $E(PRE)$. By the definition of the edge set of a slice, there must exist flow edges $v \longrightarrow_f u$ and $w \longrightarrow_f u$ in $E(PRE')$; by case (2), these edges must occur in both $E(PRE)$ and $E(PRE')$ (implying that u, v, $w \in V(PRE)$, $V(PRE')$).

Because G_M was created by a graph union, e must occur in $E(G_A / AP_{A,\,Base})$, $E(G_B / AP_{B,\,Base})$, or both. Without loss of generality, assume that $e \in E(G_A / AP_{A,\,Base})$, so that $e \in E(G_A)$.

The slice operation is a backward closure, so $e \notin E(PRE)$ and $u \in V(PRE)$ imply $e \notin E(G_{Base})$; by the definition of $AP_{A,\,Base}$, we conclude that $u \in V(AP_{A,\,Base})$.

This yields a contradiction analogous to the one that arose in case (2): by the lemma, $u \in V(AP_{A,\,Base})$ implies that $u \notin V(G_{Base} / \overline{AP}_{A,\,Base})$. However, $G_{Base} / (\overline{AP}_{A,\,Base} \cap \overline{AP}_{B,\,Base}) \subseteq G_{Base} / \overline{AP}_{A,\,Base}$, which means that $u \notin V(PRE)$.

We conclude that PRE and PRE' cannot differ; that is, even if variants A and B interfere with respect to base program *Base*, the slice $G_{Base} / (\overline{AP}_{A,\,Base} \cap \overline{AP}_{B,\,Base})$ is preserved in G_M. □

The base program, the two variants, and the merged program share common slices; thus, the next matter to address is the relationship between the execution behaviors of two programs when they share a common slice. An immediate consequence of the Slicing Theorem is that two programs that share a slice agree on all variables for which there are final-use vertices in the slice.

SLICING COROLLARY. *Let P and Q be two programs that share a slice with respect to a set of program points S (i.e. P / S is isomorphic to Q / S). Then, for any initial state σ on which both P and Q halt, the final states produced by P and Q agree on all variables for which there are final-use vertices in S.*

PROOF. Immediate from the Slicing Theorem. □

Using the Slicing Corollary, the definition of the merged graph G_M, and Preserved Behavior Lemma, we can now characterize the execution behavior of the integrated program in terms of the behaviors of the base program and the two variants. In particular, the Integration Theorem asserts that the program M that results from successfully integrating variants A and B with base program *Base* exhibits the changed behaviors of both A and B (with respect to *Base*) as well as the unchanged behavior of all three.

THEOREM. (INTEGRATION THEOREM). *If A and B are two variants of Base for which integration succeeds (and produces program M), then for any initial state σ on which A, B, and Base all halt, (1) M halts on σ, (2) if x is a variable defined in the final state of A for which the final states of A and Base disagree, then the final state of M agrees with the final state of A on x, (3) if y is a variable defined in the final state of B for which the final states of B and Base disagree, then the final state of M agrees with the final state of B on y, and (4) if z is a variable on which the final states of A, B, and Base agree, then the final state of M agrees with the final state of Base on z.*

Note that there may be some variables which do not fall into the categories of (2), (3), and (4) above.

PROOF. We use $A/AP_{A,\,Base}$, $B/AP_{B,\,Base}$, and $Base/(\overline{AP}_{A,\,Base} \cap \overline{AP}_{B,\,Base})$ to denote programs whose program dependence graphs are $G_A/AP_{A,\,Base}$, $G_B/AP_{B,\,Base}$, and $G_{Base}/(\overline{AP}_{A,\,Base} \cap \overline{AP}_{B,\,Base})$, respectively.

Since the integration succeeds, $G_A/AP_{A,\,Base} = G_M/AP_{A,\,Base}$ and $G_B/AP_{B,\,Base} = G_M/AP_{B,\,Base}$. By the Preserved Behavior Lemma, $G_{Base}/(\overline{AP}_{A,\,Base} \cap \overline{AP}_{B,\,Base}) = G_M/(\overline{AP}_{A,\,Base} \cap \overline{AP}_{B,\,Base})$. Therefore, $G_M = G_A/AP_{A,\,Base} \cup G_B/AP_{B,\,Base} \cup G_{Base}/(\overline{AP}_{A,\,Base} \cap \overline{AP}_{B,\,Base}) = G_M/AP_{A,\,Base} \cup G_M/AP_{B,\,Base} \cup G_M/(\overline{AP}_{A,\,Base} \cap \overline{AP}_{B,\,Base})$.

Since A halts on σ, by the Slicing Theorem $A/AP_{A,\,Base}$ also halts on σ. Similarly, $B/AP_{B,\,Base}$ and $Base/(\overline{AP}_{A,\,Base} \cap \overline{AP}_{B,\,Base})$ halt on σ, as well. Note that $A/AP_{A,\,Base}$ $B/AP_{B,\,Base}$, and $Base/(\overline{AP}_{A,\,Base} \cap \overline{AP}_{B,\,Base})$ are programs whose program dependence graphs are $G_M/AP_{A,\,Base}$, $G_M/AP_{B,\,Base}$, and $G_M/(\overline{AP}_{A,\,Base} \cap \overline{AP}_{B,\,Base})$, respectively. Since $A/AP_{A,\,Base}$ $B/AP_{B,\,Base}$, and $Base/(\overline{AP}_{A,\,Base} \cap \overline{AP}_{B,\,Base})$ halt on σ, by the Corollary of the Termination Theorem, M halts on σ. (This demonstrates clause (1) of the Integration Theorem.)

Let x be a variable defined in the final state of A on which the final states of A and *Base* disagree. Let v be the final-use vertex of x, so $v \in V(G_A)$. By the Slicing Theorem, $G_{Base}/v \neq G_A/v$. Therefore, $v \in AP_{A,\,Base}$. Since $v \in AP_{A,\,Base}$ and $G_A/AP_{A,\,Base} = G_M/AP_{A,\,Base}$, by the Slicing Corollary the final state of M agrees with the final state of A on x. (This demonstrates clause (2) of the Integration Theorem.) Similarly, if y is a variable defined in the final state of B on which the final states of B and *Base* disagree, then the final state of M agrees with the final state of B on y. (This demonstrates clause (3) of the Integration Theorem.)

What remains to be shown is that if z is a variable on which the final states of A, B, and *Base* agree, then the final state of M agrees with the final state of *Base* on z. Let v be the final-use vertex of z. If $v \in AP_{A,\,Base}$, since $G_A/AP_{A,\,Base} = G_M/AP_{A,\,Base}$, by the Slicing Corollary, the final state of M agrees with the final state of A on z, which means the final state of M agrees with the final state of *Base* on z. Similarly, if $v \in AP_{B,\,Base}$, since $G_B/AP_{B,\,Base} = G_M/AP_{B,\,Base}$, by the Slicing Corollary, the final state of M agrees with the final state of B on x, which means the final state of M agrees with the final state of *Base* on x. Finally, since v is a final-use vertex of G_M, if $v \notin AP_{A,\,Base}$ and $v \notin AP_{B,\,Base}$, then $v \in \overline{AP}_{A,\,Base} \cap \overline{AP}_{B,\,Base}$. Because $G_{Base}/(\overline{AP}_{A,\,Base} \cap \overline{AP}_{B,\,Base}) = G_M/(\overline{AP}_{A,\,Base} \cap \overline{AP}_{B,\,Base})$, by the Slicing Corollary the final state of M agrees with the final state of *Base* on x. (This demonstrates clause (4) of the Integration Theorem.) □

7. RELATION TO PREVIOUS WORK

This paper continues the study of program dependence graphs and program semantics begun in [6]. The Equivalence Theorem proven in [6] addresses the relationship between isomorphic PDGs; the Equivalence Theorem shows that if the program dependence graphs of two programs are isomorphic then the programs are strongly equivalent (*i.e.* given the same initial state, either both diverge or both halt in the same final state). By contrast, the Slicing and Termination Theorems proven in this paper concern *non-isomorphic* PDGs. For example, the Slicing Theorem relates the execution behavior of a program to the execution behavior of one of its slices; it demonstrates that a slice captures a *portion* of a program's behavior (in the sense that, for any initial state on which the program halts, the program and the slice compute the same sequence of values for each element of the slice).

Earlier work on program slicing includes [15], [16], and [10]. All of this previous work imposes the condition that a slice be a program whose statements are in the same relative order that they are in the original program. The notion of a slice presented in this paper is a more liberal one: the slice of a program P with respect to a set of program points S is any program Q whose PDG is isomorphic to G_P/S. In particular, the relative order of *Q*'s statements is not necessarily the same as in *P*. This generalization is justified by the Equivalence Theorem from [6] together with the Feasibility Lemma from this paper.

A version of the Slicing Theorem (for the more limited notion of slice described above) was demonstrated in [15]; it applies to a different algorithm for extracting slices, based on solving a sequence of dataflow problems, rather than the one studied here, which is based on walking backwards over the edges of a program dependence graph.

The idea of extracting a program slice by walking backwards over dependence edges appears in [10],[6] although that paper gives no justification for the operation. The Feasibility Lemma proven in this paper, (which demonstrates that any slice of a feasible program dependence graph is itself a feasible graph), together with the Slicing Theorem provide the necessary syntactic and semantic justifications, respectively, for this method of extracting slices.

When *def-order dependences* are used in program dependence graphs, larger classes of strongly equivalent programs have isomorphic program dependence graphs than when *output dependences* are used [6]. Thus, our use of def-order dependences in place of the more usual output dependences increases the number of programs that are an acceptable outcome for a given slicing operation. (For instance, the following programs are examples of two strongly equivalent programs whose PDGs are isomorphic if def-order dependences are used, but are not isomorphic if output dependences are used:

program *Main*	**program** *Main*
$x := 0;$	$x := 1;$
$a := x;$	$b := x;$
$x := 1;$	$x := 0;$
$b := x;$	$a := x;$
$x := 2$	$x := 2$
end(a, b, x)	**end**(a, b, x)

The program dependence graphs for these programs have the same (empty) set of def-order dependences, but have different sets of output dependences.)

This paper also provides semantic justification for the program-integration algorithm presented in [512]; the integration algorithm either merges two variant programs with a base program or determines that the variants incorporate interfering changes. In Section 6, the Slicing and Termination Theorems are used to show that the program that results from a successful integration operation preserves the changed behaviors of the two variants as well as the unchanged behavior of the base program.

Both this paper and [6] make use of the programming language's operational semantics to relate program dependence graphs to program semantics. Both papers start with the definition of program dependence graph as a given; the theorems that are proven, the Equivalence Theorem and the Slicing Theorem, relate certain program dependence graphs via the standard (operational) semantics of the programs that correspond to these graphs.

A different approach, using the language's denotational semantics, has been developed by Felleisen and Cartwright in [2]. Through a sequence of steps that restructure the language's semantic equations, Felleisen and Cartwright decompose the meaning function into two subsidiary functions: one that constructs (a structure similar to) a program dependence graph, and one that interprets these graphs. Their proof of the

[6]As pointed out earlier, the kind of slicing that can be performed using a program dependence graph is more restricted than the kind that can be performed with Weiser's algorithm [16]: rather than permitting a program to be sliced with respect to program point *p* and an *arbitrary* variable, a slice must be taken with respect to a variable that is defined at or used at *p*.

transformations' correctness leads directly to an analogue of the Equivalence Theorem.

It should be pointed out that there is a difference in philosophy between this paper and [2] concerning program termination. The semantics developed by Felleisen and Cartwright (as well as the corresponding dependence graph that they derive) incorporates the notion that an ". . . assignment makes no sense if a previous assignment to the variable aborts" [2]. This is in contrast with the semantics of slices obtained with our definitions of program dependence graphs and program slicing; because a diverging computation may be "sliced out" of a program, a program slice may converge on some initial states for which the original program diverges. This is illustrated by the following example (repeated from the beginning of Section 4):

program *Main*
  $x := 1$;
  while true do
    $x := x + 1$
  od;
  $x := 0$
end(x)
	program *Main*
	  $x := 0$
	end(x)

The program shown above on the left always diverges, whereas the program on the right, obtained by slicing the left-hand-side program with respect to variable *x* at the program's **end** statement, always converges. For this phenomenon to be captured with techniques like the ones used by Felleisen and Cartwright, a different "demand semantics" than the one presented in [2] is required.

REFERENCES

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

2. Felleisen, M. and Cartwright, R., "A semantic basis for program dependence graphs," Extended abstract, Department of Computer Science, Rice University, Houston, TX (December 1987).

3. 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).

4. Horwitz, S., Prins, J., and Reps, T., "Integrating non-interfering versions of programs," Technical Report 690, Department of Computer Sciences, University of Wisconsin—Madison (March 1987).

5. 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).

6. 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).

7. 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).

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

9. 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).

10. 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).

11. Reps, T. and Teitelbaum, T., "The Synthesizer Generator," pp. 42-48 in *Proceeding of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments,* , Pittsburgh, PA (April 23-25, 1984).

12. Reps, T. and Horwitz, S., "Semantics-based program integration," pp. 1-20 in *Proceedings of the Second European Symposium on Programming,* (Nancy, Franc e, March 21-24, 1988), *Lecture Notes in Computer Science,* Vol. 300, ed. H. Ganzinger,Springer-Verlag, New York, NY (1988).

13. Reps, T. and Teitelbaum, T., *The Synthesizer Generator: A system for constructing language-based editors,* Springer-Verlag, New York, NY (1988).

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

15. Weiser, M., “Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method.,” Ph.D. dissertation, University of Michigan, Ann Arbor, MI (1979). As reported by personal communication from M. Weiser, July 1988.

16. Weiser, M., “Program slicing,” *IEEE Transactions on Software Engineering* **SE-10**(4) pp. 352-357 (July 1984).