THE SEMANTICS OF PROGRAM SLICING

by

Thomas Reps and Wuu Yang

Computer Sciences Technical Report #777

June 1988
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 proof of 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 proof of 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 \( p \). The value of \( x \) at program point \( p \) is directly affected by assignments to \( x \) that reach \( p \) and by the loops and conditionals that enclose \( p \). A slice is determined from the closure of the directly-affects relation.

Program slicing, originally defined by Weiser as a data flow analysis problem [12], 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]; 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

This work was supported in part by the National Science Foundation under grants DCR-8552602 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.
modifications to $P$ interfere.

Ostenstein and Ostenstein proposed the use of program dependence graphs (PDGs) to represent programs in software development environments and pointed out how well-suited PDGs are for program slicing [10]. Program dependence graphs are an extension to the dependence graphs originally introduced by Kuck as an intermediate program representation for performing optimizing, vectorizing, and parallelizing transformations [7–9, 11]. 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.

In this paper, we address 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 proof of 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 proof of 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.

In [5] an algorithm is presented for integrating several related, but different variants of a base program (or determining whether 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) determining 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). The Slicing and Termination Theorems are used to prove a theorem, the Program 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 Program Integration Theorem asserts 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 Program 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:

```plaintext
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 $Assign$, $While$, $IfThen Else$, $Stmt List$, and $Program$. The five operators of the abstract syntax have the following definitions:

- **Assign**: $id \times exp \rightarrow stmt$
- **While**: $exp \times stmt\_list \rightarrow stmt$
- **IfThen Else**: $exp \times stmt\_list \times stmt\_list \rightarrow stmt$
- **Stmt List**: $stmt^* \rightarrow stmt\_list$
- **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], 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

\(^1\)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:
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 \textit{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 \textit{initial definition of} \( x \). This vertex represents an assignment to \( x \) from the initial state. The vertex is labeled "\( x := \text{InitialState}(x) \)."

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

The edges of \( G_P \) represent \textit{dependencies} between program components. An edge represents either a \textit{control dependence} or a \textit{data dependence}. Control dependence edges are labeled either \textit{true} or \textit{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 \( v_1 \rightarrow_c v_2 \), 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 \textit{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 \textit{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 \( v_1 \rightarrow_c v_2 \) is labeled \textit{true}; if \( v_1 \) is the predicate of a conditional statement, the edge \( v_1 \rightarrow_c v_2 \) is labeled \textit{true} or \textit{false} according to whether \( v_2 \) occurs in the \textit{then} branch or the \textit{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, program dependence graphs contain two kinds of data-dependence edges, representing \textit{flow dependences} and \textit{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:

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

4) Given a directed graph \( A = (V, E) \), a \textit{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 \((v_i, v_{i+1}) \in E\).

5) Given a directed graph \( A = (V, E) \) and a set of vertices \( V' \subseteq V \), the \textit{projection} of \( A \) onto \( V' \) is the graph \( (V', E') \), where \( E' = \{(v, w) \mid v, w \in V' \text{ and there exists a path } [v = v_1, v_2, ..., v_k = w] \text{ such that } v_2, ..., v_{k-1} \in V'\} \). (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 \textit{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.
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 \( v_1 \rightarrow_f v_2 \).

Flow dependences are further classified as loop independent or loop carried. A flow dependence \( v_1 \rightarrow_f v_2 \) is carried by loop \( L \), denoted by \( v_1 \rightarrow_{\mathcal{L}(L)} v_2 \), 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 \( v_1 \rightarrow_f v_2 \) is loop independent, denoted by \( v_1 \rightarrow_{\mathcal{L}(L)} v_2 \), 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 \( v_1 \rightarrow_{\mathcal{L}(L)} v_2 \) and \( v_1 \rightarrow_{\mathcal{L}(L)} v_2 \).

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 \( v_1 \rightarrow_f v_3 \) and \( v_2 \rightarrow_f v_3 \).

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 \( v_1 \rightarrow_{\mathcal{L}(L)} v_2 \).

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]).

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:
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.

Definition. Two programs $P$ and $Q$ are strongly equivalent iff for any state $\sigma$, either $P$ and $Q$ both diverge when initiated on $\sigma$ 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): $V(G / s) = \{ w \mid w \in V(G) \land w \rightarrow^{*,s}_{\epsilon,f} s \}$. 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 $v \rightarrow_{do(u)} w$ 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 define:

$$E(G / S) = \{(v \rightarrow_{\epsilon} w) \mid (v \rightarrow_{\epsilon} w) \in E(G) \land v, w \in V(G / S)\} \cup \{(v \rightarrow_{e} w) \mid (v \rightarrow_{e} w) \in E(G) \land v, w \in V(G / S)\} \cup \{(v \rightarrow_{do(u)} w) \mid (v \rightarrow_{do(u)} w) \in E(G) \land u, v, w \in V(G / S)\}$$

**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 $\bigcup_{i} (G / s_i) = G / \bigcup_{i} s_i$.***

---

**program Main**

$x := 1$

while $x < 11$ do

$x := x + 1$

end

end($x$)

---

**Figure 2.** The graph that results from slicing the program dependence graph from Figure 1 with respect to the final-use vertex for $x$. 
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 \( u \rightarrow w \in \bigcup_i E(G / s_i) \) we have \( u \rightarrow w \in E(G / s_i) \) for some \( i \). Because \( \{s_i\} \subseteq \bigcup_i s_i \), \( u \rightarrow w \in E(G / \bigcup_i s_i) \), so \( \bigcup_i E(G / s_i) \subseteq E(G / \bigcup_i s_i) \).

(b) For each control or flow edge \( u \rightarrow_{c,f} w \in E(G / \bigcup_i s_i) \) 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 \( u \rightarrow_{c,f} w \) we must have \( u \in V(G / s_i) \) as well, so \( u \rightarrow_{c,f} w \in E(G / s_i) \subseteq \bigcup_i E(G / s_i) \).

By the definition of the edge set of a slice, for each def-order edge \( u \rightarrow_{d^o} w \in E(G / \bigcup_i s_i) \) 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 \( u \rightarrow_f t \) and \( w \rightarrow_f t \) we have \( t,u,w \in V(G / s_i) \), so \( u \rightarrow_{d^o} w \in E(G / s_i) \subseteq \bigcup_i E(G / s_i) \).

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 \( \bigcup_i (G / s_i) = G / \bigcup_i s_i \). \( \square \)

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. \( G \) = \( G_P \)).

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

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

\(^3\)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."
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 $a \rightarrow_f b$ in $E(G_Q)$, where $b$ is not an initial-def vertex. Since $a \rightarrow_f b \in E(G_Q)$, it must be that $a \rightarrow_f b \in E(G_P)$. 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 $V(G_Q)$.

Arguing in the other direction, suppose there is an initial-definition vertex $a$ for variable $x$ in $V(G_{Q'})$ and a flow edge $a \rightarrow_f b$ that occurs in $E(G_{Q'})$ but not in $E(G_Q)$. Since $b \in V(G_Q)$ but $a \rightarrow_f b \notin E(G_Q)$, by the definition of the edge set of a slice, $a \rightarrow_f b$ 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 $c_p \rightarrow_f b$ is in $E(G_P)$ and $b$ is in $V(G_Q)$, $c_p \rightarrow_f b$ 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 $a \rightarrow_f b$ cannot be in $E(Q')$, which is a contradiction, hence there does exist a flow edge $a \rightarrow_f b$ 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 $a \rightarrow_f b \in E(G_Q)$.

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 $a \rightarrow_x b$ is a control edge in $E(G_Q)$, then $a \rightarrow_x b$ 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 $P$, $a \rightarrow_x b$ is in $E(G_{Q'})$, as well.

2) By the definition of the edge set of a slice, if $a \rightarrow_f b$ is a flow edge in $E(G_Q)$, then $a \rightarrow_f b$ 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, $a \rightarrow_f b$ is in $E(G_{Q'})$.

3) By the definition of the edge set of a slice, if $a \rightarrow_{def(c)} b$ is a def-order edge in $E(G_Q)$, then there are flow edges $a \rightarrow_f c$ and $b \rightarrow_f c$ in $E(G_Q)$. From the argument in (2), the edges $a \rightarrow_f c$ and $b \rightarrow_f c$ 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, $a \rightarrow_{def(c)} b$ 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 $a \rightarrow_x b$ 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, $a \rightarrow_x b$ is a control edge in $E(G_P)$ and, by the definition of the edge set of a slice, the control edge $a \rightarrow_x b$ is a member of $E(G_Q)$.

2) Suppose $a \rightarrow_f b$ is a flow edge that occurs in $E(G_{Q'})$ but not in $E(G_Q)$. Since $a, b \in V(G_{Q'})$ but $a \rightarrow_f b \notin E(G_Q)$, by the definition of the edge set of a slice, $a \rightarrow_f b$ 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 \( c_p \rightarrow_f b \) is in \( E(G_P) \) and \( b \) is in \( V(G_Q) \), \( c_p \rightarrow_f y \) 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 \( a \rightarrow_f b \) cannot be in \( E(Q') \), which is a contradiction; therefore, \( a \rightarrow_f b \) is a flow edge in \( E(G_Q) \).

3) If \( a \rightarrow_{\phi(c)} b \) is a def-order edge in \( E(G_Q) \), then \( a \rightarrow_f c \) and \( b \rightarrow_f c \) are in \( E(G_Q') \). From the argument in (2), the edges \( a \rightarrow_f c \) and \( b \rightarrow_f c \) are in \( E(G_P) \) and \( E(G_Q) \), as well. Because \( a \rightarrow_{\phi(c)} b \) 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 \( a \rightarrow_{\phi(c)} b \) 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, \( a \rightarrow_{\phi(c)} b \) 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. \( \square \)

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 \( G_Q = (G_P / S) \), and write this as \( P / S \).

4. 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
  x := 1;
  while true do
    x := x + 1
  od;
  x := 0
end(x)
```

```
program Main
  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 \textit{sequence} of values are assigned to the target variable; (2) for a predicate the same \textit{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 $\sigma$ is a state on which $P$ halts, then for any state $\sigma'$ that agrees with $\sigma$ on all variables for which there are initial-definition vertices in $G_Q$: (1) $Q$ halts on $\sigma'$, (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 a 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 $G_P = G_Q$. 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 $G_P = G_Q$. Let $T$ be a subtree of $P$ and $U$ be the corresponding subtree of $Q$. If $\sigma$ is a state on which $T$ halts, then for any state $\sigma'$ that agrees with $\sigma$ on their imported variables, (1) $U$ halts on $\sigma'$, (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 \( \sigma \) and \( \sigma' \), 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-syntactic operator that appears at the root of \( T \).

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

**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 \( \mathcal{X} \) a function of variables \( \{ y_j \} \); \( \text{Imp} \) is either \( \{ y_j \} \) or \( \{ y_j \} \cup \{ \mathcal{X} \} \) (\( \text{Imp} \) is \( \{ y_j \} \cup \{ \mathcal{X} \} \) when \( T \) is the target of a def-order edge). Since the value of the \( \mathcal{X} \) is a function of \( \{ y_j \} \); and \( \{ y_j \} \subseteq \text{Imp} \), evaluating \( \mathcal{X} \) in both \( \sigma_i \) and \( \sigma_i' \) yields the same value because they agree on \( \text{Imp} \). 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) \( T \) 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 \( \text{Imp} \_\text{exp} \) and \( \text{Exp} \_\text{exp} \) to denote the imported and exported variables of the \( \text{exp} \) component, respectively; we use \( \text{Imp} \_\text{stmt} \_\text{list} \) and \( \text{Exp} \_\text{stmt} \_\text{list} \) to denote the imported and exported variables of the \( \text{stmt} \_\text{list} \) component, respectively. We use \( \sigma_i \) and \( \sigma_i' \) 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 \( \text{Imp} \), \( \sigma_i \) and \( \sigma_i' \), respectively.

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

First, we show that \( \text{Imp} = \text{Imp} \_\text{exp} \cup \text{Imp} \_\text{stmt} \_\text{list} \). It is clear that we could have written this with \( \subseteq \), noting that \( \text{Imp} \_\text{stmt} \_\text{list} \) 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 \( v \in \text{Imp} \).

Let \( \sigma_i \) and \( \sigma_i' \) be states that agree on \( \text{Imp} \). Therefore they agree on \( \text{Imp} \_\text{exp} \). Evaluating the condition (the \( \text{exp} \) component) in \( \sigma_i \) and \( \sigma_i' \) 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 \( \sigma_i \) and \( \sigma_i' \), respectively.

Now suppose the condition evaluates to true. Let \( \sigma_i \) and \( \sigma_i' \) be states that agree on \( \text{Imp} \); therefore they agree on \( \text{Imp} \_\text{stmt} \_\text{list} \). Now \( T \_\text{stmt} \_\text{list} \) and \( U \_\text{stmt} \_\text{list} \) are corresponding subtrees. By the induction hypothesis of the structural induction, \( T \_\text{stmt} \_\text{list} \) and \( U \_\text{stmt} \_\text{list} \) compute the same sequence of values at each corresponding program point of the \( \text{stmt} \_\text{list} \) in the \( i \)-th iteration and the final states, \( \sigma_{i+1} \) and \( \sigma_{i+1}' \), agree on \( \text{Exp} \_\text{stmt} \_\text{list} \).

---

\(^4\)If \( x \in \text{Exp} \), then \( U \) contains an assignment \( a \) to \( x \) with an outgoing flow edge \( a \rightarrow_{\text{f}} b \). Because the loop may execute zero times, the assignment to \( x \) must be the target of a def-order edge \( \rightarrow_{\text{d}(a)} d \), hence \( x \in \text{Imp} \).
We need to show that $\sigma_{i+1}$ and $\sigma_{i+1}'$ agree on $\text{Imp}$. If $\sigma_{i+1}$ and $\sigma_{i+1}'$ do not also agree on $\text{Imp}$, then let $x \in \text{Imp}$ be a variable on which they disagree (so $x \notin \text{Exp}_{\text{stmt list}}$). Now, by assumption, $\sigma_i$ and $\sigma_i'$ agree on $\text{Imp}$; therefore, at least one of the two executions of $T_{\text{stmt list}}$ and $U_{\text{stmt list}}$, respectively, executed an assignment statement $\alpha$ that assigned a value to $x$ and reached the end of the $\text{stmt list}$. There are two cases to consider:

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

2. The other possibility is that $x \in \text{Imp}$ because there is an incoming def-order edge $\cdots \rightarrow_{d(c)} d$ in the $\text{stmt list}$. However, this implies that there is an outgoing flow edge $a \rightarrow_f c$ from $T_{\text{stmt list}}$ or $U_{\text{stmt list}}$, depending on whether $T_{\text{stmt list}}$ or $U_{\text{stmt list}}$ executed $a$. In either case, however, $x \in \text{Exp}_{\text{stmt list}}$, which contradicts our previous assumption.

We conclude that $\sigma_{i+1}$ and $\sigma_{i+1}'$ agree on $\text{Imp}$. 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 $\text{IfThenElse}$ operator. Note that $T$ and $U$ have the same $\text{exp}$ component. Because $\sigma_i$ and $\sigma_i'$ agree on $\text{Imp}$ and $\text{Imp}_{\text{exp}} \subseteq \text{Imp}$, evaluating the condition (the $\text{exp}$ component) in $\sigma_i$ and $\sigma_i'$ yields the same value; thus, $T$ and $U$ compute the same (sequence of) values at the control predicate of the $\text{IfThenElse}$ statement. Without loss of generality, assume that the condition evaluates to true.

We use $T_{\text{true}}$, $T_{\text{false}}$, $U_{\text{true}}$ and $U_{\text{false}}$ to denote the respective branches of $T$ and $U$. Note that $T_{\text{true}}$ and $U_{\text{true}}$ are corresponding subtrees and $T_{\text{false}}$ and $U_{\text{false}}$ are corresponding subtrees. We use $\text{Imp}_{\text{true}}$ and $\text{Exp}_{\text{true}}$ to denote the imported and exported variables of $T_{\text{true}}$, respectively; we use $\text{Imp}_{\text{false}}$ and $\text{Exp}_{\text{false}}$ to denote the imported and exported variables of $T_{\text{false}}$, respectively.

When execution is initiated in state $\sigma_i$, $T$ terminates in $\sigma_2$; consequently $T_{\text{true}}$ also terminates in $\sigma_2$. Since $\sigma_i$ and $\sigma_i'$ agree on $\text{Imp}$ and $\text{Imp}_{\text{true}} \subseteq \text{Imp}$, $\sigma_i$ and $\sigma_i'$ agree on $\text{Imp}_{\text{true}}$. Because $T_{\text{true}}$ and $U_{\text{true}}$ are corresponding subtrees, the induction hypothesis tell us that, when execution is initiated in state $\sigma_i'$, (1) $U_{\text{true}}$ terminates in state $\sigma_2'$ (hence $U$ terminates in state $\sigma_2'$), and (2) $T_{\text{true}}$ and $U_{\text{true}}$ 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 $\text{StmtList}$ operator. Let $T_1, T_2, \ldots, T_n$ 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 $U_1, U_2, \ldots, U_n$ 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 $\pi$ is a permutation over the interval $1 \ldots n$. Let $\pi^{-1}$ denote the inverse permutation of $\pi$.

We use $\sigma_i$ and $\sigma_{\pi(i)}'$ to denote the execution states before executing $T_i$ and $U_{\pi(i)}$, respectively; we use $\text{Imp}_i$ and $\text{Exp}_i$ 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 $\text{Imp}_i$ and $\text{Exp}_i$.

The proof of this case is by induction over $i$. We want to show that for all $i$, $1 \leq i \leq n$, if $\sigma_i$ and $\sigma_i'$ agree on $\text{Imp}$ and $T$ halts on $\sigma_i$, then $\sigma_i$ and $\sigma_{\pi(i)}'$ agree on $\text{Imp}_i$ 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 $\sigma_i$ and $\sigma_{\pi(i)}'$ agree on $\text{Imp}_i$ 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 $\sigma_i \text{ and } \sigma_{\pi(i)} \text{ agree on } Imp_i$, for all $i$, $1 \leq i \leq n$.

**Base case.** $i = 1$. First we show that $\sigma_1'$ 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 $\sigma_1'$ and $\sigma_{\pi(i)}'$ do not agree on $Imp_1$, let $x \in Imp_1$ be a variable on which they disagree. The execution of the initial subsequence $U_1, U_2, \ldots, U_{\pi(i)-1}$ 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 $a \to \ldots$, 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 $T_{\pi'(k)}$ to a vertex in $T_1$. Therefore, $\pi'(k) < 1$, which is a contradiction because $T_1$ is the first immediate subtree of $T$. We conclude that $\sigma_1'$ and $\sigma_{\pi(i)}'$ agree on $Imp_1$.

Because $\sigma_1$ and $\sigma_{\pi(1)}'$ agree on $Imp_1$ and $Imp_1 \subseteq Imp_1 \subseteq Imp_1$, $\sigma_1$ and $\sigma_{\pi(1)}'$ agree on $Imp_1$. Because $\sigma_1$ and $\sigma_{\pi(1)}'$ agree on $Imp_1$ and $\sigma_{\pi(i)}'$ agree on $Imp_1$, $\sigma_1$ and $\sigma_{\pi(i)}'$ 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 $\sigma_1$ and $\sigma_{\pi(i)}'$, respectively.

**Induction step.** The induction hypothesis is: if $\sigma_1$ and $\sigma_{\pi(1)}'$ agree on $Imp_1$ and $T$ halts on $\sigma_1$, then $\sigma_j$ 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 $1 \leq j \leq i$. Thus, if $\hat{\sigma}_1$ and $\hat{\sigma}_{\pi(i)}'$ 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 $T_1, T_2, \ldots, T_i$, 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 $T_1, T_2, \ldots, T_i$. Since $\hat{\sigma}_1$ and $\hat{\sigma}_i$ agree on $Imp$, $\hat{\sigma}_i$ and $\hat{\sigma}_1$ agree on $x$. Thus, $\hat{\sigma}_{i+1}$ and $\hat{\sigma}_{\pi(i+1)}'$ 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 $U_1, U_2, \ldots, U_{\pi(i+1)}$ 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 $a \to \ldots$, 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 $T_{\pi'(k)}$ to a vertex in $T_{i+1}$. Therefore, $\pi'(k) < i+1$. Because $U_k$ has an assignment statement, $a$, to $x$, $T_{\pi'(k)}$ has a corresponding assignment statement to $x$, which contradicts a previous assumption that there is no assignment statement to $x$ in the initial subsequence $T_1, T_2, \ldots, T_i$.

2. Suppose there are assignment statements that assign to $x$ in the initial subsequence $T_1, T_2, \ldots, T_i$. Let $m$ be the largest number in $1, 2, \ldots, i$ 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 $T_{m+1}, T_{m+2}, \ldots, T_i$. 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 \( m \leq i \), by the induction hypothesis, \( \hat{\sigma}_m \) and \( \hat{\sigma}_{n(m)} \) agree on \( \text{Imp}_m \). Because \( \hat{\sigma}_m \) and \( \hat{\sigma}_{n(m)} \) agree on \( \text{Imp}_m \) and \( T \) and \( U \) are corresponding subtrees, by the Equivalence Lemma, the execution states after executing \( T \) and \( U \) agree on \( \text{Exp}_m \), and hence they agree on \( x \). Thus, \( \hat{\sigma}_{i+1} \) and \( \hat{\sigma}_{n(m)+1} \) agree on \( x \).

Because \( \hat{\sigma}_{i+1} \) and \( \hat{\sigma}_{n(i+1)+1} \) do not agree on \( x \), \( \hat{\sigma}_{n(m)+1} \) and \( \hat{\sigma}_{n(i+1)+1} \) do not agree on \( x \). Thus the execution of the subsequence \( U \hat{\sigma}_{n(m)+1} U \hat{\sigma}_{m+1} \ldots \hat{\sigma}_{i+1} \) executed an assignment statement to \( x \) and reached the beginning of \( U \). Let the assignment statement \( a \) be in \( U_k \), where \( \pi(m) \leq k \leq \pi(i+1) \). Since \( x \) is an imported variable of \( U \), \( U \hat{\sigma}_{i+1} \) has an incoming loop-independent flow edge or an incoming def-order edge \( a \rightarrow \ldots \), 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 \( T \hat{\sigma}_{n(k)} \) 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 \( U \hat{\sigma}_{n(m)} \) and \( U_k \) reach \( U \hat{\sigma}_{n(i+1)} \). Hence \( U_k \) has an incoming def-order edge whose source is in \( U \hat{\sigma}_{n(m)} \). Since \( T \) and \( U \) have isomorphic program dependence graphs, \( T \hat{\sigma}_{n(k)} \) 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 \( T \hat{\sigma}_{n(k)} \) are corresponding subtrees, there is an assignment statement to \( x \) in \( T \hat{\sigma}_{n(k)} \), which contradicts a previous assumption that \( m \) is the largest number in \( 1, 2, \ldots, i \) such that \( T_m \) contains an assignment statement to \( x \).

We conclude that \( \hat{\sigma}_{i+1} \) and \( \hat{\sigma}_{n(k)+1} \) agree on \( \text{Imp}_{i+1} \). By the induction hypothesis of the structural induction, \( T \) and \( U \) 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 \( \text{Imp} = \text{Imp}_{\text{stmt, inst}} \), the proposition that \( T \) and \( U \) compute the same sequence of values at each corresponding program point follows directly from the induction hypothesis. \( \Box \)

### 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 \( G_P = G_Q \). If \( \sigma \) is a state on which \( P \) halts, then for any state \( \sigma' \) that agrees with \( \sigma \) on all variables for which there are initial-definition vertices in \( G_P \): (1) \( Q \) halts on \( \sigma' \), (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. \( \Box \)

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 $\sigma$ is a state on which $T$ halts, then (1) $U$ halts on $\sigma'$ where $\sigma$ and $\sigma'$ 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-syntactic operator that appears at the root of $T$.

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

**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 $\sigma_I$ and $\sigma_I'$ yields the same value because they agree on $Imp_U$. $Exp_U$ is either $\emptyset$ or $\{x\}$. For any combination of these possibilities, $\sigma_2$ and $\sigma_2'$ 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; $Imp_{stmt_list}$ and $Exp_{stmt_list}$ denote the imported and exported variables of $U$'s $stmt_list$ component, respectively. We use $\sigma_i$ and $\sigma_i'$ 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 $Imp_U$, $\sigma_i$ and $\sigma_i'$, 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, $\sigma_{j+1}$ and $\sigma_{j+1}'$ agree on $Exp_U$. Because for a loop $Exp_U \subseteq Imp_U$, it suffices to show that if $\sigma_I$ and $\sigma_I'$ agree on $Imp_U$ then either $T$ and $U$ halt in the states $\sigma_i$ and $\sigma_i'$, respectively, or the $i^{th}$ iterations compute the same sequence of values at each program point of $U$ and result in the states $\sigma_{i+1}$

---

3 If $x \in Exp_U$, then $U$ contains an assignment $a \rightarrow x$ with an outgoing flow edge $a \rightarrow T b$. Because the loop may execute zero times, the assignment to $x$ must be the target of a def-order edge $\rightarrow_{def}(a)x$, hence $x \in Imp_U$. 

and $\sigma_i' = 1$ that agree on $Imp_U$.

First, we show that $Imp_U = Imp_{exp} \cup Imp_{U_{sum}}$. It is clear that we could have written this with $\subseteq$, noting that $Imp_{U_{sum}}$ 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 flow edge to $t$, which implies that $v \in Imp_U$.

Let $\sigma_i$ and $\sigma_i'$ be states that agree on $Imp_U$. Therefore they agree on $Imp_{exp}$. Evaluating the condition (the $exp$ component of $U$) in $\sigma_i$ and $\sigma_i'$ 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 $\sigma_i$ and $\sigma_i'$, which agree on $Exp_U$.

Now suppose the condition evaluates to true. Let $\sigma_i$ and $\sigma_i'$ be states that agree on $Imp_U$; therefore they agree on $Imp_{U_{sum}}$. Now $T_{stmt\_list}$ and $U_{stmt\_list}$ are corresponding subtrees. Since $T$ halts on $\sigma_i$, $T_{stmt\_list}$ also halts on $\sigma_i$. By the induction hypothesis, (1) $U_{stmt\_list}$ halts on $\sigma_i'$, (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, $\sigma_{i+1}$ and $\sigma_{i+1}'$, agree on $Exp_{U_{sum}}$. If $\sigma_{i+1}$ and $\sigma_{i+1}'$ do not also agree on $Imp_U$, then let $x \in Imp_U$ be a variable on which they disagree (so $x \notin Exp_{U_{sum}}$). Now, by assumption, $\sigma_i$ and $\sigma_i'$ 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 $\ldots \rightarrow_f b$ in $U$. If this were the case, then there must be a loop-carried flow edge $a \rightarrow_{L+(T)} b$ or $a \rightarrow_{L+(U)} b$, 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 $x \in Exp_{U_{sum}}$, 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 $\ldots \rightarrow_{def(c)} d$. However, this implies that there is an outgoing flow edge $a \rightarrow_f c$ 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 $x \in Exp_{U_{sum}}$, which contradicts our previous assumption.

We conclude that $\sigma_{i+1}$ and $\sigma_{i+1}'$ 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 $\sigma_{j+1}$ and $\sigma_{j+1}'$ agree on $Exp_U$.

Case 3. The operator at the root of $T$ is the IThenElse 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 $\sigma_i$ and $\sigma_i'$ yields the same value. Therefore, $T$ and $U$ compute the same (sequence of) values at the control predicate of the IThenElse 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 $\sigma_i$, $T$ terminates in $\sigma_i'$; consequently $T_{true}$ also terminates in $\sigma_i'$. Since $\sigma_i$ and $\sigma_i'$ agree on $Imp_U$ and $Imp_{U_{sum}} \subseteq Imp_U$, $\sigma_i$ and $\sigma_i'$ agree on $Imp_{U_{sum}}$. Because $T_{true}$ and $U_{true}$ are corresponding subtrees, the induction hypothesis tells us that, when execution is initiated in state $\sigma_i'$, (1) $U_{true}$ terminates in state $\sigma_i'$ (hence $U$ terminates in $\sigma_i'$), (2) $T_{true}$ and $U_{true}$ compute the same sequence
of values at each program point of \( \text{U}_\text{true} \) (hence \( T \) and \( U \) compute the same sequence of values at each program point of \( U \)), and (3) \( \sigma_2 \) and \( \sigma'_2 \) agree on \( \text{Exp}_U \).

Note that \( \text{Exp}_U = \text{Exp}_U \cap \text{Exp}_U \) so what remains to be shown is that \( \sigma_2 \) and \( \sigma'_2 \) agree on \( \text{Exp}_U \). If \( \sigma_2 \) and \( \sigma'_2 \) do not also agree on \( \text{Exp}_U \), then let \( x \in \text{Exp}_U \) be a variable on which they disagree (so \( x \notin \text{Exp}_U \)). Because \( x \in \text{Exp}_U \), 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 \( a \rightarrow_f b \)).

We must consider whether it is possible that \( x \notin \text{Imp}_U \). By assumption, \( x \notin \text{Exp}_U \). 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 \( c \rightarrow_f b \). By construction, vertex \( c \) occurs to the left of \( a \), hence \( c \rightarrow_d (b) \). We conclude that \( x \in \text{Imp}_U \).

Since \( x \in \text{Imp}_U \), \( \sigma_1 \) and \( \sigma'_1 \) agree on \( x \). Because \( \sigma_2 \) and \( \sigma'_2 \) 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 \( d \rightarrow_f b \) in either \( T \) or \( U \), depending on whether \( T \text{true} \) or \( U \text{true} \) executed the assignment to \( x \). In either case, the flow edge \( d \rightarrow_f b \) is in \( Q \) since \( b \) is in \( Q \), and hence \( d \) is in \( U \text{true} \). Therefore, \( x \in \text{Exp}_U \), which contradicts a previous assumption. We conclude that \( \sigma_2 \) and \( \sigma'_2 \) agree on \( \text{Exp}_U \). This, together with the fact that \( \sigma_2 \) and \( \sigma'_2 \) agree on \( \text{Exp}_U \), means that they agree on \( \text{Exp}_U \).

Case 4. The operator at the root of \( T \) is the StmtList operator. Let \( T_1, T_2, \ldots, T_n \) denote the immediate subtrees of \( T \) and \( U_1, U_2, \ldots, U_n \) denote the corresponding subtrees of \( U \). (Note that some of the \( U_i \) may be the tree \( \text{Null} \).) We use \( \sigma_i \) and \( \sigma'_i \) to denote the execution states before executing \( T_i \) and \( U_i \), respectively; we use \( \text{Imp} \_U_i \) and \( \text{Exp} \_U_i \) to denote the imported and exported variables, respectively, of \( U_i \); and we use \( \text{Imp} \_U_i \) and \( \text{Exp} \_U_i \) to denote the imported and exported variables, respectively, of the initial subsequence \( U_1, U_2, \ldots, U_i \). (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 \( i \), \( 1 \leq i \leq n \), if \( \sigma_1 \) is a state on which \( T \) halts and \( \sigma_1 \) and \( \sigma'_1 \) agree on \( \text{Imp}_U_1 \), then \( T_1 \) and \( U_1 \) terminate in \( \sigma_i+1 \) and \( \sigma'_i+1 \), respectively, and \( T_{1..i} \) and \( U_{1..i} \) compute the same sequence of values at each program point of \( U_{1..i} \) and \( \sigma_i+1 \) and \( \sigma'_i+1 \) agree on \( \text{Exp}_U_{1..i} \).

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

Induction step. The induction hypothesis is: if \( \sigma_1 \) and \( \sigma'_1 \) agree on \( \text{Imp}_U_1 \), and \( T \) halts on \( \sigma_1 \), then \( T_{1..i} \) and \( U_{1..i} \) terminate in \( \sigma_i+1 \) and \( \sigma'_i+1 \), respectively, and \( T_{1..i} \) and \( U_{1..i} \) compute the same sequence of values at each program point of \( U_{1..i} \). Since \( \sigma_1 \) and \( \sigma'_1 \) are arbitrary states that agree on \( \text{Imp}_U_1 \) and \( T \) halts on \( \hat{\sigma}_1 \), we need to show that \( T_{1..i} \) and \( U_{1..i} \) terminate in \( \sigma_{i+1} \) and \( \sigma'_{i+1} \), 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 \( \sigma_{i+1} \) and \( \sigma'_{i+1} \) agree on \( \text{Exp}_U_{1..i+1} \).

Note that \( \text{Imp}_U_{1..i+1} \subseteq \text{Imp}_U_{1..i} \), which means that \( \sigma_1 \) and \( \hat{\sigma}'_1 \) agree on \( \text{Imp}_U_{1..i+1} \), and thus, by the induction hypothesis, \( T_{1..i} \) and \( U_{1..i} \) terminate in \( \sigma_{i+1} \) and \( \sigma'_{i+1} \), respectively, and \( T_{1..i+1} \) and \( U_{1..i+1} \) compute the
same sequence of values at each program point of \( U_{1,i} \), and (3) \( \hat{\sigma}_{i+1} \) and \( \hat{\sigma}_{i+1}' \) agree on \( \text{Exp}_{U_{i+1}} \).

First, we must show that \( \hat{\sigma}_{i+1} \) and \( \hat{\sigma}_{i+1}' \) agree on \( \text{Imp}_{U_{i+1}} \). Any variable \( x \in \text{Imp}_{U_{i+1}} \) on which \( \hat{\sigma}_{i+1} \) and \( \hat{\sigma}_{i+1}' \) disagree must be in \( \text{Imp}_{U_{i+1}} \) (if not, \( x \) would be in \( \text{Exp}_{U_{i+1}} \) on which \( \hat{\sigma}_{i+1} \) and \( \hat{\sigma}_{i+1}' \) agree). By assumption, \( \hat{\sigma}_{i} \) and \( \hat{\sigma}_{i}' \) agree on \( \text{Imp}_{U_{i}} \); 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 \( T_{i} \) or \( U_{i} \) performed the assignment. There are now two cases to consider:

1. One possibility is that \( x \in \text{Imp}_{U_{i}} \) 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: \( a \rightarrow \gamma b \) 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 \( x \in \text{Exp}_{U_{i}} \), 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 \( x \in \text{Imp}_{U_{i}} \), because there is an incoming def-order edge, \( \ldots \rightarrow d_{\omega}(a) \rightarrow c \), to \( U_{i+1} \). However, this implies that there is an outgoing flow edge of: \( a \rightarrow \gamma d \) in \( T \) or \( U \) depending whether \( T \) or \( U \) performed the assignment, \( a \). In either case, \( a \) is in \( U \) because \( d \) is in \( U \). Therefore, this flow edge \( a \rightarrow \gamma d \) is in \( Q \). As in the previous case, this implies that \( x \in \text{Exp}_{U_{i}} \), 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 \( \text{Imp}_{U_{i+1}} \).

Because \( T \) terminates on \( \hat{\sigma}_{i} \), \( T_{i+1} \) must terminate on \( \hat{\sigma}_{i+1} \). Because \( \hat{\sigma}_{i+1} \) and \( \hat{\sigma}_{i+1}' \) agree on \( \text{Imp}_{U_{i+1}} \), 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 \( \text{Exp}_{U_{i+1}} \).

The final step is to show that \( \hat{\sigma}_{i+2} \) and \( \hat{\sigma}_{i+2}' \) agree on \( \text{Exp}_{U_{i+1}} \). Note that \( \text{Exp}_{U_{i+1}} \subseteq \text{Exp}_{U_{i+1}} \cup \text{Exp}_{U_{i}} \). Now suppose there is a variable \( x \in \text{Exp}_{U_{i+1}} \) on which \( \hat{\sigma}_{i+2} \) and \( \hat{\sigma}_{i+2}' \) disagree (in particular, \( x \notin \text{Exp}_{U_{i}} \)). Therefore, \( x \in \text{Exp}_{U_{i+1}} \). By the induction hypothesis, \( \hat{\sigma}_{i+1} \) and \( \hat{\sigma}_{i+1}' \) agree on \( \text{Exp}_{U_{i+1}} \), 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 \( x \in \text{Exp}_{U_{i+1}} \) \( \subseteq \text{Exp}_{T_{i+1}} \), there must also be an outgoing flow edge \( a \rightarrow \gamma \ldots \) from \( T_{i+1} \). Since \( U_{i+1} \) is a slice of \( T_{1..i+1} \) and \( x \in \text{Exp}_{U_{i+1}} \), therefore the flow edge \( a \rightarrow \gamma \ldots \) is in \( U_{1..i+1} \). This implies that \( x \in \text{Exp}_{U_{i+1}} \), 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 \( x \in \text{Exp}_{U_{i+1}} \) there must also be an outgoing flow edge \( a \rightarrow \gamma \ldots \) from \( U_{i+1} \). This implies that \( x \in \text{Exp}_{U_{i+1}} \), 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 \( \sigma' \), \( T \) and \( U \) compute the same sequence of values at each program point of \( U \), and \( \sigma_{n+1} \) and \( \sigma_{n+1}' \) agree on \( \text{Exp}_{U} \).

Case 5. The operator at the root of \( T \) is the Program operator. Because \( \text{Imp}_{U} = \text{Imp}_{U_{n+1}} \) and \( \text{Exp}_{U} = \text{Exp}_{U_{n+1}} \), we conclude from the induction hypothesis that \( U \) terminates on \( \sigma' \), \( T \) and \( U \) compute the same sequence of values at each program point of \( U \), and \( \sigma_{2} \) and \( \sigma_{2}' \) agree on \( \text{Exp}_{U} \). \( \Box \)

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 \( \sigma \) is a state on which \( P \) halts, then for any state \( \sigma' \) that agrees with \( \sigma \) on all variables for which there are
initial-definition vertices in $G_Q$: (1) $Q$ halts on $\sigma'$, (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. $\square$

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 $\sigma_U$ is a state on which $U$ halts, and $\sigma_V$ is a state on which $V$ halts. Then for any state $\sigma$, where $\sigma$ and $\sigma_U$ agree on $U$'s imported variables and $\sigma$ and $\sigma_V$ agree on $V$'s imported variables, $T$ halts on $\sigma$.

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 $\sigma$ and $\sigma_U$ agree on $U$'s imported variables and $U$ halts on $\sigma_U$, by the Self-Equivalence Lemma $T$ halts on $\sigma$.

**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 $\text{Null}$ tree, then $T = V$. Similarly, if $V$ is a $\text{Null}$ tree, then $T = U$. Without loss of generality, suppose $T = U$. Because $\sigma$ and $\sigma_U$ agree on $U$'s imported variables and $U$ halts on $\sigma_U$, by the Self-Equivalence Lemma $T$ halts on $\sigma$.

Now suppose both $U$ and $V$ are not $\text{Null}$ trees. Since $U$ halts on $\sigma_U$, 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 $\sigma$ and $\sigma_V$, respectively, after exactly $j$ iterations. Because for a loop $\text{Exp}_U \subseteq \text{Imp}_U$, it suffices to show that if $\sigma$ and $\sigma_U$ agree on $U$'s imported variables and $\sigma$ and $\sigma_V$ agree on $V$'s imported variables, then either $T$, $U$, and $V$ halt in the states $\sigma$, $\sigma_U$, and $\sigma_V$, respectively, or $T$, $U$, and $V$ successfully finish one iteration and the execution states that result after one iteration of the loops ($\sigma'$, $\sigma'_U$, and $\sigma'_V$, respectively) are ones such that $\sigma'$ and $\sigma'_U$ agree on $U$'s imported variables and $\sigma'$ and $\sigma'_V$ agree on $V$'s imported variables.

We use $T_{\text{stmt_list}}$, $U_{\text{stmt_list}}$, and $V_{\text{stmt_list}}$ to denote the stmt_list components of $T$, $U$, and $V$, respectively. Note that $T_{\text{stmt_list}}$, $U_{\text{stmt_list}}$, and $V_{\text{stmt_list}}$ are corresponding subtrees of $P$, $P / X$, and $P / Y$, respectively.
Because $\sigma$ and $\sigma_U$ agree on $U$'s imported variables, evaluating the control predicates in $\sigma$ and $\sigma_U$ yields the same value. Because $\sigma$ and $\sigma_V$ agree on $V$'s imported variables, evaluating the control predicates in $\sigma$ and $\sigma_V$ yields the same value. If the control predicate evaluates to false, then $T$, $U$, and $V$ halt in the states $\sigma$, $\sigma_U$, and $\sigma_V$, respectively.

Now suppose the control predicate evaluates to true. Because $\sigma$ and $\sigma_U$ agree on $U$'s imported variables and the imported variables of $U_{\text{stmt list}}$ are a subset of $U$'s imported variables, $\sigma$ and $\sigma_U$ agree on the imported variables of $U_{\text{stmt list}}$. Similarly, $\sigma$ and $\sigma_V$ agree on the imported variables of $V_{\text{stmt list}}$. Note that $T_{\text{stmt list}}$, $U_{\text{stmt list}}$, and $V_{\text{stmt list}}$ are corresponding subtrees of $P$, $P/X$, and $P/Y$, respectively. Because $U_{\text{stmt list}}$ and $V_{\text{stmt list}}$ halt on $\sigma_U$ and $\sigma_V$, respectively, by the induction hypothesis, $T_{\text{stmt list}}$ halts on $\sigma$. Therefore, $T$, $U$, and $V$ successfully finish one iteration. Let $\sigma'$, $\sigma_U'$, and $\sigma_V'$ denote the execution states of $T$, $U$, and $V$ after one iteration of the loop, respectively. By the Subtree Slicing Lemma, $\sigma'$ and $\sigma_U'$ agree on $U$'s exported variables and $\sigma'$ and $\sigma_V'$ agree on $V$'s exported variables. By the same argument as in the proof of the Subtree Slicing Lemma, Case 2, $\sigma'$ and $\sigma_U'$ agree on $U$'s imported variables. Similarly, $\sigma'$ and $\sigma_V'$ agree on $V$'s imported variables. We conclude that $T$, $U$, and $V$ halt on $\sigma$, $\sigma_U$, and $\sigma_V$, 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 $\text{Null}$ tree, then $T = V$. Similarly, if $V$ is a $\text{Null}$ tree, then $T = U$. Without loss of generality, suppose $T = U$. Because $\sigma$ and $\sigma_U$ agree on $U$'s imported variables and $U$ halts on $\sigma_U$, by the Self-Equivalence Lemma $T$ halts on $\sigma$.

Now suppose both $U$ and $V$ are not $\text{Null}$ trees. Because $\sigma$ and $\sigma_U$ agree on $U$'s imported variables, evaluating the control predicates in $\sigma$ and $\sigma_U$ yields the same value. Because $\sigma$ and $\sigma_U$ agree on $V$'s imported variables, evaluating the control predicates in $\sigma$ and $\sigma_V$ yields the same value. Without loss of generality, we may assume the control predicate evaluates to true.

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

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

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

The proof of this case is by induction over the initial subsequences of $T$. We want to show that for all $i$, $1 \leq i \leq n$, if $U_1, \ldots, i$ and $V_1, \ldots, i$ halt on $\sigma_U$ and $\sigma_V$, respectively, then $T_1, \ldots, i$ halts on $\sigma$ where $\sigma$ and $\sigma_U$ agree on the imported variables of $U_1, \ldots, i$, and $\sigma$ and $\sigma_V$ agree on the imported variables of $V_1, \ldots, i$.

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

Induction step. The induction hypothesis is: if $U_1, \ldots, i$ and $V_1, \ldots, i$ halt on $\sigma_U$ and $\sigma_V$, respectively, then $T_1, \ldots, i$ halts on $\sigma$ where $\sigma$ and $\sigma_U$ agree on the imported variables of $U_1, \ldots, i$, and $\sigma$ and $\sigma_V$ agree on the imported variables of $V_1, \ldots, i$. Thus, if $\sigma_U$ and $\sigma_V$ are arbitrary states on which $U_1, \ldots, i+1$ and $V_1, \ldots, i+1$ 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 $\hat{\sigma}'_V$ 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 $\sigma$.

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 $\sigma$. □

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 $\sigma$, then $P$ halts on $\sigma$ 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 $\sigma$, then $P$ halts on $\sigma$ 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 $\sigma$, by the Subtree Termination Lemma, $P/(X \cup Y)$ halts on $\sigma$. 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 $\sigma$, $P$ halts on $\sigma$. □

6. 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 [5]. 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.

We now describe the steps of the integration algorithm. The first step 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

\[ AP_{A, Base} = \{ v | \forall v \in V(G_A) \land (G_{Base}(v) \neq G_A(v)) \} \]

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 is 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 no interference of this kind if G_M / AP_{A, Base} = G_A / AP_{A, Base} and G_M / AP_{B, Base} = 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 will be returned as the result of the integration operation.

6.1. The Integration Theorem

Using the Slicing Theorem and the definition of the merged graph G_M, we now show the following theorem, which 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 $s$ on which $A$, $B$, and Base all halt, (1) $M$ halts on $s$, (2) if $x$ is a variable on 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 on which the final states of $B$ and Base disagree, then the final state of $M$ agrees with the final state of $B$ on $y$, 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 $G_{Base} / (AP_{A,Base} \cap AP_{B,Base})$. 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 $G_{Base} / (AP_{A,Base} \cap AP_{B,Base}) = G_M / (AP_{A,Base} \cap AP_{B,Base})$.

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

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

**Proof.** From the definition, $AP_{A,Base} = \{ v \in V(A) \mid (G_{Base} / v \neq G_A / v) \}$. So $AP_{A,Base} = \{ v \in V(A) \mid (G_{Base} / v = G_A / v) \}$. Using the Decomposition Lemma, we have:

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

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 $w \notin G_{Base} / AP_{A,Base}$. □

**LEMMA (PREERVED BEHAVIOR LEMMA). Let**

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

Then $G_{Base} / (AP_{A,Base} \cap AP_{B,Base}) = G_M / (AP_{A,Base} \cap AP_{B,Base})$.

**Proof.** Let $PRE = G_{Base} / (AP_{A,Base} \cap AP_{B,Base})$ and $PRE' = G_M / (AP_{A,Base} \cap AP_{B,Base})$. 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, $AP_{A,Base} \cap AP_{B,Base}$, 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, $AP_{A,Base} \cap AP_{B,Base}$, $PRE'$ can only contain an additional vertex $v$ if there is an additional control or flow edge $v \rightarrow_{c,f} w$ 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 $AP_{A,Base} \cap AP_{B,Base}$, 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 $e : v \rightarrow w$,
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 \( w \notin V(G_{Base} / AP_{A,Base}) \). However, \( G_{Base} / (AP_{A,Base} \cap AP_{B,Base}) \subseteq G_{Base} / AP_{A,Base} \), which means that \( w \notin V(PRE) \).

Case 3. \( PRE' \) contains an additional def-order edge. Suppose \( E(PRE') \) contains a def-order edge \( e : v \rightarrow_{do} w \) that does not occur in \( E(PRE) \). By the definition of the edge set of a slice, there must exist flow edges \( v \rightarrow_f u \) and \( w \rightarrow_f u \) 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 \( u \notin V(G_{Base} / AP_{A,Base}) \). However, \( G_{Base} / (AP_{A,Base} \cap AP_{B,Base}) \subseteq G_{Base} / AP_{A,Base} \), 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 \( G_{Base} / (AP_{A,Base} \cap AP_{B,Base}) \) is preserved in \( G_M \). \( \square \)

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. \( (P / S) = (Q / S) \)). Then, for any initial state \( \sigma \) 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 \( V(P / S) \).

**PROOF.** Immediate from the Slicing Theorem. \( \square \)

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 \( \sigma \) on which \( A, B, \) and \( Base \) all halt, (1) \( M \) halts on \( \sigma \), (2) if \( x \) is a variable on 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 on which the final states of \( B \) and \( Base \) disagree, then the final state of \( M \) agrees with the final state of \( B \) on \( y \), 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 \( \sigma \), by the Slicing Theorem \( A / AP_{A,Base} \) also halts on \( \sigma \). Similarly, \( B / AP_{B,Base} \) and \( Base / (\overline{AP}_{A,Base} \cap \overline{AP}_{B,Base}) \) halt on \( \sigma \), 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 \( \sigma \), by the Corollary of the Termination Theorem, \( M \) halts on \( \sigma \). (This demonstrates clause (1) of the Integration Theorem.)

Let \( x \) be a variable on which the final states of \( A \) and \( Base \) disagree. Let \( v \) be the final-use vertex of \( x \). 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 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.)

Let \( z \) be 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, if \( v \in AP_{A,Base} \) and \( v \in AP_{B,Base} \), then \( v \in AP_{A,Base} \cap AP_{B,Base} \). Because \( G_{Base} / (AP_{A,Base} \cap AP_{B,Base}) = G_M / (AP_{A,Base} \cap AP_{B,Base}) \), 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. By contrast, the Slicing Theorem proven in this paper concerns the relationship between two non-isomorphic PDGs. 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).

Program slicing was first defined by Weiser as a data flow analysis problem [12]. The idea to extract program slices by taking the backward closure of a dependence graph with respect to a set of vertices is
due to Ottenstein and Ottenstein [10], although they gave 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 extracting slices via a backward closure over dependence edges.

In both [12] and [10], a condition is imposed that requires the program produced as the result of slicing to have its statements ordered in the same relative order as they occur 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.

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
  x := 0;
  a := x;
  x := 1;
  b := x;
  x := 2;
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 [5]; 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 Theorem is used to show that the integrated program that results from a successful integration operation preserves the changed behaviors of the two variants and 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 transformations' correctness leads directly to an analogue of the Equivalence Theorem.

---

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 [12]: 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 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)
```

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