February 24, 1993

Graph theoretic foundations of program slicing and integration

Arun Lakhotia

The Center for Advanced Computer Studies
University of Southwestern Louisiana
Lafayette, LA 70504
(318) 231-6766, -5791 (Fax)
arun@cacs.usl.edu

Available as CACS TR-91-5-5

Revision history: 3/20/92, 3/16/92, 12/2/91, 11/25/91

1 Abstract

This paper generalizes program slicing algorithms originally defined over representations of programs to operate over directed graphs. Doing so provides a uniform framework to model Weiser's and Ottenstein & Ottenstein's approaches to program slicing as abstract mathematical operations transparent of any concerns of a program's structure or its semantics. This transparency helps us in a) deriving calculational style proofs of algebraic properties of slices, b) making more general assertions about these properties than those previously established, and c) generalizing Weiser's slicing criterion to allow union of statements.

The two program integration algorithms due to Reps and Horwitz, Prins, & Reps use program slicing as an elementary operation and are generalized to integrate directed graphs. These algorithms can therefore be used to integrate versions of any artifact that may be represented as graphs, for instance versions of specification and design of software systems.

2 Introduction

A *slice* of a program with respect to a program point p and variable u consists of all statements of the program that may affect the value of u at point p. Program slicing, the technique of extracting a program's slice was introduced by Weiser [Wei84]. He showed that program slicing aided in the debugging of programs. Later works have used program slices in measuring module cohesion and assisting in software maintenance.

Later works [KL88, AH90] introduce the notion of *dynamic slice*- the set of statements that affect the value of a variable at a particular 'instance' of a program point during the execution of a specific input. In comparison a slice defined by Weiser is termed as a *static slice*. In this paper, except in Section 8, we restrict our attention to static program slices. Henceforth, unless explicitly stated, a program slice implies a static program slice.

There are two approaches to program slicing, one due to Weiser [Wei84] and the other due to Ottenstein & Ottenstein [OO84]; referred to here as Weiser's slice and O-O slice, respectively. The approaches differ on their *slicing criterion* - factor on which a slice is performed - and the technique for

Table 1 A comparison of various slicing algorithms. All but Weiser's algorithm identify the set of statements in a slice by performing a reachability analysis of their representation graph based on the slicing criterion. The modified Weiser's algorithm of Section 3 uses the same slicing criterion as Weiser's algorithm and produces similar results by performing reachability analysis.

Algorithm	Scope procedure	Representation	Slicing criterion	Technique
Weiser's [Wei84]	multiple	Flowgraph	Statement, Variable	incremental flow analysis
Ottenstein & Ottenstein [HPR88a]	single	Program Dependence graph	Statement	reachability analysis
Horwitz, Reps, Binkley [HRB90]	multiple	System dependence graph	Statement	reachability analysis
modified Weiser's section 3	single	extended PDG	Statement, Variable	reachability analysis

detecting statements belonging to a slice, as shown in Table 1. The specific slicing algorithms[Gal90, HRB90, OO84, OT89, Wei84] further differ on whether they process single procedure or multiple procedure programs[*] and the internal representation they use for programs.

An interesting application of program slicing is the problem of integrating multiple versions of programs. This problem, formalized by Horwitz, Prins, and Reps [HPR88a], may informally be stated as follows. Let *A* and *B* be two programs that are variants of a third program *Base*. Let *M* be a program such that a) it "preserves the changed behavior" of both *A* and *B* wrt to *Base* and b) it "preserves the preserved behaviors" of *Base* in both *A* and *B*. The problem of program integration is to find *M* given *A*, *B*, and *Base*. If such a program does not exist, *A* and *B* are said to *interfere* with *Base* and the algorithm should detect it.

The need for program integration commonly arises in scenarios where two persons may perform simultaneous modifications to a program and these modifications are needed to be merged such that the conditions stated above are satisfied. Horwitz, Prins, and Reps used program slicing to identify the preserved and changed behaviors between programs. There are three reported algorithms for program integration: HPR algorithm [HPR88a], Reps' algorithm [Rep90], Yang's algorithm [Yan90]. Figure 1 gives a schematic diagram comparing the internal representations used by these algorithms.

As is obvious, the proof of correctness of the slicing and integration algorithms and of properties of their results depend very strongly on the internal representation of their programs and the operations they use to produce the results [HPR88a, Rep90, Wei84]. Since the algorithms use different representations and/or operations, these proofs need to be established afresh for every combination.

In this paper, we define a *modified* Weiser's algorithm that has the same slicing criterion as Weiser's algorithm but uses reachability analysis to detect statements belonging to a slice. This implies that both O-O and modified Weiser's slicing algorithms perform reachability analysis to detect statements in their slice. They however differ the representation of the program on which they operate and their slicing criterion. This leads us to abstract reachability analysis, the technique for detecting statements in a slice, as an operation on directed graphs. We call this *graph slicing*.

[*] We use the phrases single procedure slice and multiple procedure slice to mean intraprocedural slice and interprocedural slice, respectively. Due to the similarity in their spellings, the latter phrases require extra caution by the authors and the readers alike.

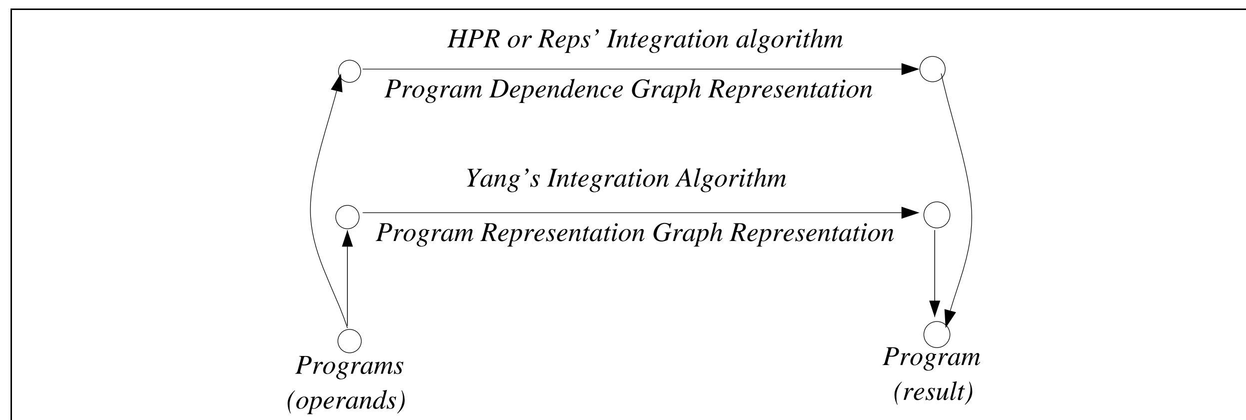

Figure 1 Schematic view of HPR, Reps', and Yang's program integration algorithms. The algorithms translate the operand program into their respective internal representation, integrate the operands in these internal representation, and then create the program that corresponds to this internal representation.

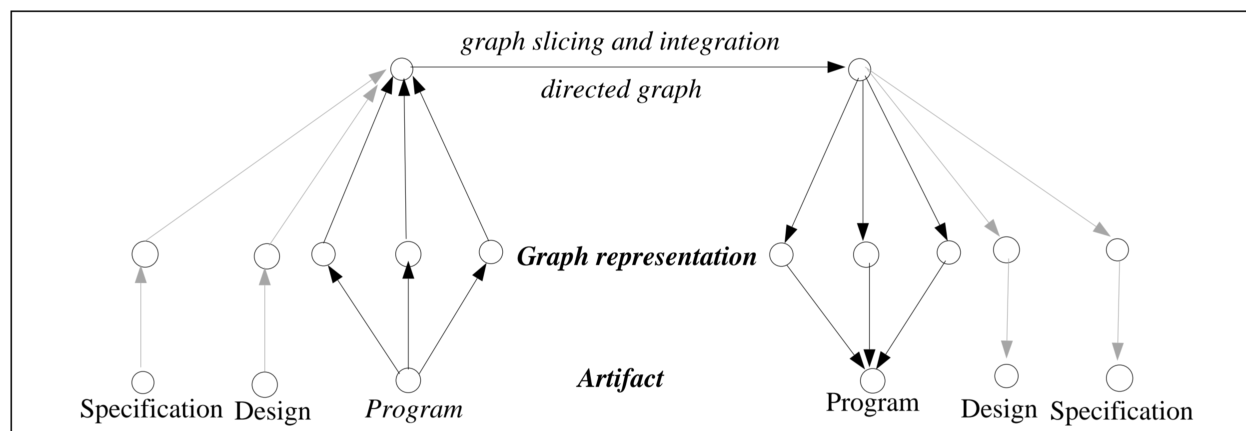

Figure 2 Schematic view of the benefits of abstracting slicing and integration operations as operations over directed graphs. The HPR and Reps' program integration algorithm can be used to integrate programs in graph representations other than PDG. Analogously, artifacts from other domains (designs, specifications) may also be sliced and integrated with meaningful results. The constraints on the representation or the meaning of the resulting artifacts are discussed in Section 9.

We further note that HPR and Reps' integration algorithms, but not Yang's algorithm, are defined over program dependence graph representation of programs using program slicing, graph-theoretic, and set-theoretic operations to perform the operation. We define analogous algorithm on directed graphs using graph-slicing instead. We call this operation *graph integration.*

This generalization of program slicing and program integration to operations over graphs has the benefit that it allows the investigation of properties of these operations independent of the context of their usage. This leads to isolating the properties of these algorithms into two categories:

1. those that can be derived solely from graph-theoretic reasoning, referred to herein as *syntactic properties*, and

2. those that depend on the interpretation associated to the graph representations, referred to herein as *semantic properties*.

This implies that the syntactic properties for program slicing and integration algorithms can be derived from the properties of analogous graph operations. The proof of these properties for every combination of representations and/or operations can be extrapolated from the proofs of the corresponding properties for graphs.

We model Weiser and O-O program slicing algorithms using graph slicing. We discover that the property of 'union' of slices - a syntactic property, for modified Weiser's slices can be generalized further than what was investigated by Weiser. Loosely speaking, Weiser had found that the union of slices at the same statement but different variables is the same as the slice at that statement and the two variables. We find that the union of slices at different statements but the same variables is also the same as the slice over the union of the slicing criterion.

The properties that relate the meaning of the resulting program to that of the operand programs for both integration and slicing algorithms are classified as semantic properties. These properties, intrinsically, need to be established for every combination of interpretation associated to a language and the relations encoded by the graph representations. Reps and Yang [RY89] use operational interpretation of programs and formulate the meaning of a program preserved in its slices as the *slicing theorem*. Similarly, they establish the meaning of the program generated by the HPR integration algorithm in terms of the operand programs and stated it as the *integration theorem*.

Our generalization gives some interesting insights into the integration operation. It splits the *integration theorem* into two parts: the syntactic integration theorem and the semantic integration theorem. The first theorem states relation between the structures of the operands and results of integration as graph-theoretic expressions whereas the second theorem relates the meanings of the operands and results. Since our directed graphs have no specific interpretation we state the second theorem in terms of an abstract interpretation. We further show that the semantic integration theorem can be proved for *any* interpretation that satisfies a *slicing axiom*.

One implication of this generalization is that if a multiple procedure slicing algorithm can be abstracted in terms of *graph slicing* and satisfies the slicing theorem then it can be used with the HPR and Reps' program integration algorithms, currently defined for integrating single procedure programs, to integrate programs for multiple procedure programs. The Horwitz, Reps, and Binkley's [HRB90] multiple procedure slicing algorithm unfortunately can not be abstracted as graph slicing and hence the possibility of its use for program integration with HPR or Reps' integration algorithm has to be investigated explicitly (as in [Bin91]); it does not follow from our findings.

There are other interesting implications of our generalization. These are discussed in Section 9 and outlined in Figure 2.

The rest of the paper is organized as follows. In the next section we present background information. This includes notations used in this paper, definition of program dependence graph, HPR and modified Weiser's program slicing algorithm, and HPR program integration algorithm. Section 4 defines graph slicing and presents some properties of these slices. Section 5 models the HPR and modified Weiser's slicing algorithms using the abstraction developed in Section 4. Section 6 defines graph integration and Section 7 outlines the requisite *slicing axiom* that should hold to derive the semantic integration theorem. Section 9 presents our conclusions of the implications of our generalization.

3 Background

Notations

$G = (V, E)$ is a directed graph if $E \subseteq V \times V$, where V is a set of vertices of the graph and E its edges. We use the notation $\nu(G)$ to denote V and $\epsilon(G)$ to denote E. Further, for elements belonging to the set of edges we use the notation $v \rightarrow u$ to denote the coordinate pair (v, u). The notation $v \rightarrow^+ u \in \epsilon(G)$ denotes that either $v \rightarrow u \in \epsilon(G)$ or $\exists n_1, n_2, ..., n_i \in \nu(G), i > 0$ such $v \rightarrow n_1 \in \epsilon(G) \wedge ... \wedge n_i \rightarrow u \in \epsilon(G)$.

The graph $G_1 = (V_1, E_1)$ is a *subgraph* of G if $V_1 \subseteq V$ and $E_1 = E \cap V_1 \times V_1$.

V^2 is used as a shorthand for $V \times V$ and is extrapolated to V^n.

If $\theta : \mathcal{D}_1 \times \mathcal{D}_2 \rightarrow \mathcal{D}_3$ is a binary operator then $\vec{\theta} : \mathcal{D}_1^n \times \mathcal{D}_2^n \rightarrow \mathcal{D}_3^n$ extends it over *n*-tuple and is defined as follows. Let $T = <t_1, t_2, \ \ldots \ , t_n> \in \mathcal{D}_1^n$ and $U = <u_1, u_2, \ \ldots \ , u_n> \in \mathcal{D}_2^n$ be n-tuples. $T \vec{\theta} U = <t_1 \theta u_1, t_2 \theta u_2, \ \ldots \ , t_n \theta u_n>$. Thus the set theoretic operators $\cup$ and $\cap$ extended to $\vec{\cup}$ and $\vec{\cap}$ respectively.

The ith element of an *n*-tuple is referred as $T \uparrow_i$. When used with a set of *n*-tuple it gives the set of ith elements of all its elements.

Programs

We restrict our study to programs written in a simple programming language consisting of *assignment*, *if-then-else*, *while*, *read*, and *write* statements with the conventionally accepted syntax. It also has an *end* statement that appears at the end of a program and may have a list of variables. This is a special type of output statement and is used to list the variables whose values are of interest at the end of the program. The language also has statements to declare the start of a program and to declare the variables local to a program. Only scalar variables are permitted. The procedure and declaration statements are considered *non-executable statements* and are not significant for slicing and integration. Other statements are called *executable*. This language consists of some common features of languages used Horwitz, Prins, and Reps [HPR88a] and Weiser [Wei84] for performing single procedure slices. Restricting ourselves to this language enables us to compare slicing and integration algorithms from different sources.

The program slicing algorithms are guided by data and control dependences between the statements of a program which may be gathered using flow analysis [ASU86, Hec77]. Two sets of variables def and use are associated with the node for each statement, except the procedure statement. The first consists of all the variables whose values may be changed by that statement and latter the set of variables whose values it uses. The two sets are defined as follows:

"i contains $y := f(e_1, ..., e_n)$" then $def(i) = \{y\}$ and $use(i) = \{x \mid x$ 'occurs in some' $e_j, 1 \leq j \leq n\}$.
"i contains **read** $x_1, ..., x_n$" then $def(i) = \{x_1, ..., x_n\}$ and $use(i) = \{\}$.
"i contains **print** $e_1, ..., e_n$" then $def(i) = \{\}$ and $use(i) = \{x \mid x$ 'occurs in some' $e_j, 1 \leq j \leq n\}$.
"i contains **if** $p(e_1, ..., e_n)$ **then** ..." then $use(i) = \{x \mid x$ 'occurs in some' $e_j, 1 \leq j \leq n\}$, $def(i) = \{\}$.
"i contains **while** $p(e_1, ..., e_n)$ **do** ..." then $use(i) = \{x \mid x$ 'occurs in some' $e_j, 1 \leq j \leq n\}$, $def(i) = \{\}$.

"i contains **end**$(x_1, .., x_n)$" then $use(i) = \{x_1, .., x_n\}$ and $def(i) = \{\}$.

A variable is said to be *referred* at a statement if it is either defined or used there.

In a flowgraph [Hec77], a path $(i, n_1, ..., n_m, j)$, $m \geq 0$, containing no defs of v in nodes $n_1, ..., n_m$ is said be *definition-clear* wrt *v* from node *i* to node *j*.

A definition *d* of a variable *v* at node *x* is said to *reach* node *y* iff *d* is in *def(x)* and there is a definition-clear path wrt *v* from node *x* to node *y*. The definition at node *x* is said to reach the node *y*. With every vertex of a flowgraph we associate another set $rd(i)$ which is the set of all the vertices containing definitions that reach the vertex i.

Program dependence graph

Ottenstein and Ottenstein [OO84] define a program slice using the PDG representation of programs. Program slicing algorithms using their approach employ some variation of this graph. HPR and Reps' program integration algorithms operate on PDG representation of programs. This subsection presents a brief sketch of program dependence graphs.

The PDG representation used by [OO84] is a variation of a theme introduced in [KMC72]. The definition of program dependence graph presented here is paraphrased from [HPR88a].

The program dependence graph *G* for a program *P* is a multigraph. The vertices of a PDG represent program components and the edges represent flow and control dependences among components. There is a vertex for each assignment and control predicate in the program. In addition, PDGs include two other categories of vertices – there is a distinguished vertex called the *Entry* vertex; for each variable *x* that is used before being defined, there is a vertex labeled *x := Initialstate(x)*. The *Entry* vertex is treated as a predicate vertex (that is always true).

The edges of a PDG are divided into *control dependence* edges and *data dependence* edges. There are two kinds of data dependence edges: *flow* and *def-order* dependence edges.

The source of a control dependence edge is always a predicate vertex. There is a control edge from a predicate vertex *u* to a vertex *v* if *v* represents a program component that is immediately nested within the control construct whose predicate is represented by *u*. The control dependence edge is labeled by the truth value of the branch in which *v* occurs.

There is a flow dependence edge from vertex v_1 to vertex v_2, denoted by $v_1 \rightarrow_f v_2$, if v_1 defines a variable x and v_2 uses the variable x and control can reach v_2 after v_1 via an execution path along which there is no intervening definition of x.[†]

There is a *def-order dependence edge* from vertex v_1 to v_2 iff all of the following conditions hold: both v_1 and v_2 define the same variable; v_1 and v_2 are in the same branch of any conditional statement that encloses both of them; there exists a program component v_3 such that $v_1 \rightarrow_f v_3$ and $v_2 \rightarrow_f v_3$; and v_1 appears to the left of v_2 in the program's abstract syntax tree. A def-order edge from v_1 to v_2 is denoted by $v_1 \rightarrow_{do(v_3)} v_2$.

Figure 3 shows a program and its program dependence graph.

If P is a program its PDG is denoted by G_P.

Definition: A PDG G_P is *feasible* if there exists a program P that corresponds to it; it is *infeasible* otherwise.

[†] The flow dependence edges are further classified as loop-carried and loop-indeependent. This classification is not significant for the purposes of this paper and is omitted.

```
program
  x := 5
  y := 5
  a := 1
  while a < 5 do
     x := x + y
     y := x - y
     a := a +1
  end
end(x,y,a)
```

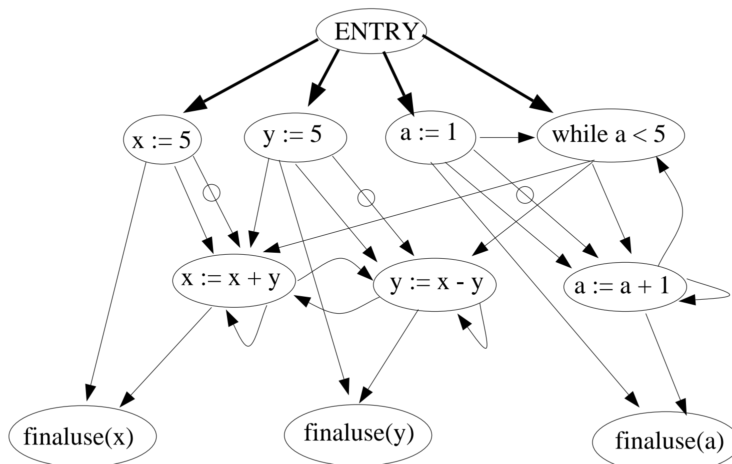

Figure 3 Program dependence graph of a sample program. The control edges are represented by boldface arrows. The truth values of these edges is not shown to keep the graph simple. Loop-carried edges are represented by solid arrows with a hash mark. Loop-independent edges are represented by solid arrows. Def-order edges are represented by solid arrows with a circle mark.

Ottenstein & Ottenstein's program slicing algorithm

Ottenstein & Ottenstein's (O-O) definition of slice [OO84] differs from Weiser's [Wei84] original definition of program slice. A program slice according to them is a set of program statements that directly or indirectly contribute to the computation performed at some program point. This *slicing criterion*, or parameters on which a program is to be sliced, according to this definition consists of a statement.

The O-O slicing algorithm operates on the program dependence graph representation of the program and hence may be stated in terms of the graph (instead of the program). For a vertex s of a PDG G_P, the *O-O slice* of G_P with respect to s, written $G_P \not\circ s$, is a graph containing all the vertices on which s has a transitive flow or control dependence:

$$\nu(G_P \not\circ s) \triangleq \{w \mid w \in \nu(G_P) \wedge (w = s \vee w \rightarrow^{+}_{c,f} s \in \epsilon(G_P))\}^{\ddagger}.$$

$$\epsilon(G_P \not\circ s) \triangleq \{(v \rightarrow_{c,f} w) \in \epsilon(G_P) \mid v, w \in \nu(G_P \not\circ s)\}$$
$$\cup\{(v \rightarrow_{do(u)} w \in \epsilon(G_P) \mid v, w, u \in \nu(G_P \not\circ s)\}.$$

Reps and Yang [RY89] show the following semantic properties of program slices computed by O-O algorithm.

1. (Feasibility Lemma). *For any program P, if G_Q is a slice of G_P then G_Q is a feasible PDG.*
2. (Slicing Theorem). *Let Q be a slice of a program P with respect to a set of vertices. If σ is a state on which P halts, then for any state σ' that agrees with σ on all variables for which there are initial definition vertices in G_Q:*

   a. *Q halts on σ'*
   b. *P and Q compute the same sequence of values at each program point of Q, and*
   c. *the final states agree on all variables for which there are final-use vertices in G_Q.*

‡ Ottenstein & Ottenstein did not express the slicing algorithm like this. This formalization was done by Horwitz, Prins, and Reps [HPR88a]. They defined $\nu(G_P \not\circ s) \triangleq \{w \mid w \in \nu(G_P) \wedge w \rightarrow^{*}_{c,f} s \in \epsilon(G_P)\}$. We believe this definition is not precise since for s to be included in the slice it requires an edge $s \rightarrow_{c,f} s \in \epsilon(G_P)$.

Horwitz, Prins, and Reps generalized O-O's slicing criterion s to consist of a *set* of statements and defined the resulting slice by replacing $\nu(G_P \not\phi\, s)$ by:

$$\nu(G_P \not\phi\, s) \triangleq \{w \mid w \in \nu(G_P) \wedge (w \in s \vee (\exists u \in s \cdot w \rightarrow^{+}_{c,f} u \in \epsilon(G_P)))\}$$

With this definition they showed that: $\vec{\cup}_i(G_P \not\phi\, s_i) = G_P \not\phi \cup_i s_i$.

We term the slicing criterion consisting of a single statement as *unit slicing criterion*.

Weiser's program slice

Weiser's slicing algorithm differs from Ottenstein and Ottenstein's in two respects:

1. its slicing criterion consists of a 2–tuple, <*statement, variable*>, and
2. it performs *incremental* data and control flow analysis of the program to detect the statements in the slice.

The implication of the first difference is that the set of slicing criterion on which a program may be sliced using Weiser's algorithm is the cross-product of the set of statements and the set of variables in the program. This set is larger than the set of statements, the set of unit slicing criterion for O-O slicing algorithm.

The second difference makes Weiser's slicing algorithm computationally more expensive than O-O's. This is because the PDG of a program can be computed using a constant number of passes of the program where as the number of passes of the program required by Weiser's algorithm depends on the nesting depth of the statement on which the slice is performed.

Agrawal and Horgan [AH90] have observed that a program's slice using Weiser's slicing criterion, say <*n,var*>, and PDG representation may be performed by "by finding all reaching definitions of *var* at node *n* and traversing the program dependence graph at these nodes". The statements in the set of vertices visited during the traversal constitute the desired slices.

The above statement, though not quite precise, gives the gist of the relationship. It may be stated more precisely as:

Definition: Let *P* be a program, *n* and *x* be a statement and a variable, respectively, in *P*, and G_P be its PDG:

$$P \not\psi {<} s, x {>} \triangleq P \not\phi\, V \ \textit{where}\ \textbf{if}\ x \in use(s) \cup def(s)\ \textbf{then}\ V = \{s\}$$
$$\textbf{else} V = \{v_i \mid x \in def(v_i) \wedge v_i \in rd(s)\}.$$

We call this algorithm *modified Weiser's algorithm*. Since a PDG does not represent the relationships *use*, *def*, and *rd*, a program slice cannot be defined completely in terms of the program representation when using a PDG. In order to be able to do so we define an *extended PDG* in Section 5 as a PDG extended to contain these relations.

Weiser also extended his slicing criterion to allow *set* of variables instead of a single variable. This can be incorporated in the above definition appropriately. Weiser showed that for a set of variables X, $\vec{\cup}_{x \in X} P \not\psi {<} s, x {>} = P \not\psi {<} s, X {>}$.

Program integration problem

Definition[HPR88a, RY89]: Informally, *integration of programs* A and B with respect to *program* $Base$ is the *program* M such that M preserves the:

a. *changes* in the meaning of A with respect to $Base$,
b. *changes* in meaning of B with respect to $Base$, and
c. *similarity* in the meanings of *both* A and B with respect to $Base$.

This may be stated more formally as follows.

Definition *[RY89]: A program M is the integration of programs A and B, two variants of a program $Base$ if for any initial state σ on which A, B, and $Base$ all halt, (1) M halts on σ, (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 state of A, B, and $Base$ agree, then the final state of M agrees with the final state of $Base$ on z.*

Definition: Two *programs* A and B *interfere* with respect to $Base$ if there does not exist a program that integrates the two.

The HPR and Reps' *program* integration algorithms operate on PDG representation of the programs. A schematic view of their functioning is shown in Figure 1. They first create PDGs for the operand programs, integrate the PDGs using their respective algorithms, and create a program corresponding to the resulting PDG.

HPR program integration algorithm

The HPR program integration algorithm implements the following expression.

$G_M = (G_A \phi AP_{A,Base}) \vec{\cup} (G_B \phi AP_{B,Base}) \vec{\cup} (G_{Base} \phi PP_{Base,A,B})$

where $AP_{X,Base} \triangleq \{v \mid v \in \nu(G_X) \wedge (G_{Base} \phi v) \neq (G_X \phi v)\}$ and

$PP_{Base,X,Y} \triangleq \{v \mid v \in \nu(G_{Base}) \wedge (G_{Base} \phi v) = (G_X \phi v) = (G_Y \phi v)\}$.

The program M corresponding to the PDG G_M, if it exists, is the integration of programs A and B with respect to $Base$, if A and B are non-interfering.

Informally, $AP_{X,Base}$ is the set of *affected points*, the set of statements of program X whose meaning is different from that in $Base$ and $PP_{Base,X,Y}$ is the set of *preserved points*, or the statements of $Base$ that have the same meaning in $Base$, X, and Y.

That the changed meanings of programs A and B is preserved in G_M is ensured if A and B do not interfere. The two programs do not interfere if 1) $G_M \phi PP_{A,Base} = G_A \phi AP_{A,Base}$ and $G_M \phi AP_{B,Base} = G_B \phi AP_{B,Base}$ and 2) G_M is a feasible PDG. These two tests for interference are referred to by Horwitz et. al. Type I and Type II interference tests, respectively.

Reps and Yang [RY89] proved that the HPR algorithm integrates non-interfering programs.

For the sake of brevity we do not present Reps' program integration algorithm [Rep90]. In Section 6 we state how this algorithm may be generalized to operate over directed graphs.

4 Unreachable subgraphs and graph slices

This section defines graph slices and some properties of slices of a graph.

The subgraph G_1 is *unreachable* from the rest of G, or simply G_1 is unreachable subgraph, if $E \cap (\overline{V_1} \times V_1) = \Phi$, where $\overline{V_1} \triangleq V - V_1$. Otherwise G_1 is *reachable*.

Theorem 4.1 *The set of all unreachable subgraphs of a graph is closed under vector intersection and union .*

Proof:[§] Let $G = (V, E)$, $G_1 = (V_1, E_1)$, and $G_2 = (V_2, E_2)$ be directed graphs such that G_1 is an unreachable subgraph, (i.e. $V_1 \subseteq V$, $E_1 = E \cap (V_1 \times V_1)$, $E \cap (\overline{V_1} \times V_1) = \Phi$) and G_2 is an unreachable subgraph, (i.e. $V_2 \subseteq V$, $E_2 = E \cap (V_2 \times V_2)$, and $E \cap (\overline{V_2} \times V_2) = \Phi$).

We have to prove that a) $G_1 \vec{\cup} G_2$ is an unreachable subgraph, (i.e. $V_1 \cup V_2 \subseteq V$, $E_1 \cup E_2 = E \cap (V_1 \cup V_2) \times (V_1 \cup V_2)$, and $E \cap \overline{(V_1 \cup V_2)} \times (V_1 \cup V_2) = \Phi$)
and b) $G_1 \vec{\cap} G_2$ is an unreachable subgraph, (i.e.$V_1 \cap V_2 \subseteq V$, $E_1 \cap E_2 = E \cap (V_1 \cap V_2) \times (V_1 \cap V_2)$, and $E \cap \overline{(V_1 \cap V_2)} \times (V_1 \cap V_2) = \Phi$).

The results a) and b) are proved in the following discussion by proving each of the individual expressions for it as a subproof. The subproofs use the given assumptions about G_1 and G_2, set theoretic deductions, and some theorems from set theory presented in the following subsection.

Subproof: $V_1 \cup V_2 \subseteq V$. []

Subproof: $E_1 \cup E_2 = E \cap (V_1 \cup V_2) \times (V_1 \cup V_2)$.

$E_1 \cup E_2$

$= E \cap (V_1 \times V_1) \cup E \cap (V_2 \times V_2)$.

$\underline{(A \times A) \cup (B \times B) = (A \cup B) \times (A \cup B) - ((A - B) \times (B - A) \cup (B - A) \times (A - B))}.$

$= E \cap (V_1 \cup V_2) \times (V_1 \cup V_2) - (E \cap (V_1 - V_2) \times (V_2 - V_1) \cup E \cap (V_2 - V_1) \times (V_1 - V_2))$

[Subproof: $E \cap (V_1 - V_2) \times (V_2 - V_1) = \Phi$ and $E \cap (V_2 - V_1) \times (V_1 - V_2) = \Phi$.
$E \cap (V_1 - V_2) \times (V_2 - V_1)$.
$\underline{A - B \subseteq \overline{B}, A - B \subseteq A, B \subseteq D \wedge C \subseteq E = B \times C \subseteq D \times E}$
$\subseteq E \cap \overline{V_2} \times V_2$. .
$= \Phi$. The other expression is symmetric].

$= E \cap (V_1 \cup V_2) \times (V_1 \cup V_2)$. []

Subproof: $E \cap \overline{V_1 \cup V_2} \times (V_1 \cup V_2) = \Phi$

$E \cap \overline{V_1 \cup V_2} \times (V_1 \cup V_2)$

$\underline{A \times (B \cup C) = (A \times B) \cup (A \times C)}$

$= E \cap ((\overline{V_1 \cup V_2} \times V_1) \cup (\overline{V_1 \cup V_2} \times V_2))$.

$\underline{\overline{A \cup B} \subseteq \overline{A}}$

$\subseteq E \cap ((\overline{V_1} \times V_1) \cup (\overline{V_2} \times V_2))$.

$= E \cap (\overline{V_1} \times V_1) \cup E \cap (\overline{V_2} \times V_2)$.

$= \Phi$. [].

Subproof: $V_1 \cap V_2 \subseteq V$. []

§ Most assertions in this paper are proven using calculational style proof procedure, [Gri91]. Hints for justification of proof steps are written in smaller font and are underlined too. For the sake of brevity, hints for steps that may be derived either from definitions or from obvious set theoretic formulations have been omitted.

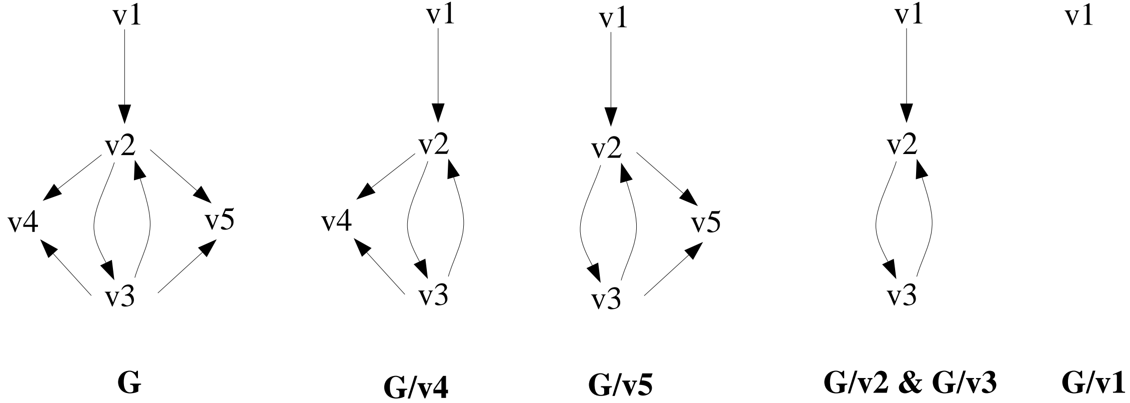

Figure 4 A graph and its slices at different vertices.

Subproof: $E_1 \cap E_2 = E \cap (V_1 \cap V_2) \times (V_1 \cap V_2)$.

$E_1 \cap E_2$.

$= E \cap (V_1 \times V_1) \cap E \cap (V_2 \times V_2)$.

$\underline{((A \times A) \cap (B \times B) = (A \cap B) \times (A \cap B).)}$

$= E \cap (V_1 \cap V_2) \times (V_1 \cap V_2)$. []

Subproof: $E \cap \overline{V_1 \cap V_2} \times (V_1 \cap V_2) = \Phi$

$E \cap \overline{V_1 \cap V_2} \times (V_1 \cap V_2)$.

$= E \cap (\overline{V_1} \cup \overline{V_2}) \times (V_1 \cap V_2)$.

$\underline{((A \cup B) \times C = (A \times C) \cup (B \times C))}$

$= E \cap (\overline{V_1} \times (V_1 \cap V_2)) \cup E \cap (\overline{V_2} \times (V_1 \cap V_2))$.

$\underline{(A \times (B \cap C) = (A \times B) \cap (A \times C))}$

$= E \cap (\overline{V_1} \times V_1) \cap (\overline{V_1} \times V_2) \cup E \cap (\overline{V_2} \times V_2) \cap (\overline{V_2} \times V_1)$.

$= \Phi$. []

Definition: Let L be the set of all unreachable subgraphs of a graph G with partial order $\sqsubseteq$ defined as: $a, b \in L$, $a \sqsubseteq b \triangleq a \vec{\sqcap} b = a$ (or equivalently, $a \sqsubseteq b \triangleq a \vec{\sqcup} b = b$). Also $a \sqsubset b \triangleq a \sqsubseteq b \land a \neq b$.

Corollary: $(L, \vec{\sqcup}, \vec{\sqcap})$ is a *lattice*. Its least element is the empty graph, ϕ, and the greatest element is G.

Definition: G' is an unreachable subgraph of G *with respect to a vertex set* s if $G' \sqsubseteq G$ and $s \subseteq \nu(G')$. It is a *minimal* unreachable subgraph with respect to s if it is such a subgraph with minimal number of vertices, i.e. if $\exists G'', G'' \sqsubseteq G$ and $s \subseteq \nu(G'')$ then $\mid \nu(G') \mid \leq \mid \nu(G'') \mid$[‖]. The notation G/s is used to denote the minimal unreachable subgraph of G wrt s and is referred to as a *graph slice* of G over s.

We next present some properties of graph slices. Their proofs are quite straight forward and are left to the reader.

Definition: $a \preceq b \triangleq a = b/\nu(a)$. a is said to be a *slice of* b.

Reps [Rep90] defines an analogous operator to relate two programs, where one program is the slice of another.

‖ $\mid S \mid$ denotes the cardinality (or size) of the set S.

Lemma 4.2 $a, b \in L,\ a \preceq b \equiv a \sqsubseteq b.$

Proof: Exercise. []

Therefore $\preceq$ also defines a partial order on L.

Theorem 4.3 The graph slice G/s may be computed as follows:

$\nu(G/s) = \{v \mid v \in s \vee (\exists s_i \cdot v \to^+ s_i \in \epsilon(G))\}$

$\epsilon(G/s) = \{u \to w \mid u, w \in \nu(G/s) \wedge u \to w \in \epsilon(G)\}$

Proof: Exercise. []

Lemma 4.4 *There is a unique graph slice with respect to a set of vertices.*

Proof: Exercise. []

Lemma 4.5 *Every unreachable subgraph is a graph slice over some set of vertices.*

Proof: Exercise. []

Lemma 4.6 $G/ \cup_i s_i = \vec{\bigcup}_i G/s_i.$

Proof: Exercise. []

Here are some interesting 'fix-point' properties of graph slices stated without proof; 'fix-point' because equations 1 and 2 have the form $G' = f(G')$ and the equality in equation 3 has the form $f(x, G) = f(x, G')$.

Lemma 4.7 *Let* $G' = G/s$.

1. $G' = G'/s.$
2. $G' = G/\nu(G').$
3. $x \in \nu(G') \to G/x = G'/x.$

Proof: Exercise. []

5 Modelling program slicing using graph slicing

We now model the O-O and Weiser's slicing algorithms in terms of graph slices. To do so we first note that the program dependence graphs and the extended dependence graphs are *not* directed graphs or multigraphs; instead a PDG is a hypergraph [Ber89] while an extended PDG is a 7–tuple as defined below.

Modelling O-O slicing algorithm

Definition: A *abstract PDG* is a 3–tuple $<V, E, D>$, where $E \subseteq V^2$ and $D \subseteq V^3$ with the constraints that: if $(v_1, v_2, v_3) \in D$ then $(v_1, v_3) \in E$ and $(v_2, v_3) \in E$. Notice that the tuple $<V, E>$ is a directed graph.

The set V above corresponds to the set of vertices, E to the set of flow and control edges, and D to the def-order edges of a PDG. The constraint over elements of D specifies the condition that if there exists a def-order edge $v_1 \to_{do(v_3)} v_2$ in the PDG of a program then there exist flow-edges $v_1 \to_f v_3$ and $v_2 \to_f v_3$ in it. This definition by no means completely models a PDG since it does not model the types

associated to vertices and edges. These types are not necessary since they are not used by O-O slicing algorithm to identify the vertices belonging to a slice.

The O-O slicing algorithm may be modelled in terms of graph slices as follows.

Definition: If $G_P = <V, E, D>$ is an abstract PDG and $G' = <V, E>$ then $G_P \not\triangleright s \triangleq <V', E', D'>$ such that $V' = \nu(G'/s)$, $E' = \epsilon(G'/s) = E \cap V'^2$, and $D' = D \cap V'^3$.

It is clear from the construction that $<V', E'>$ is a directed graph and that $D' \subseteq V'^3$. To establish that $G_P \not\triangleright s$ is a PDG we only need to prove the additional constraint on D.

Lemma 5.1 *In the definition above if $(v_1, v_2, v_3) \in D'$ then $(v_1, v_3) \in E'$ and $(v_2, v_3) \in E'$.*

Proof: $(v_1, v_2, v_3) \in D'$

$= (v_1, v_2, v_3) \in D \cap (V' \times V' \times V')$

$= (v_1, v_2, v_3) \in D$ and $v_1, v_2, v_3 \in V'$

(definition of abstract PDG)

$\Rightarrow (v_2, v_3) \in E$ and $(v_1, v_3) \in E$ and $v_1, v_2, v_3 \in V'$

$= (v_2, v_3) \in E \cap (V' \times V')$ and $(v_1, v_3) \in E \cap (V' \times V')$

$= (v_2, v_3) \in E'$ and $(v_1, v_3) \in E'$. []

We now prove that the union of O-O slice is the same as the slice over the union of their slicing criterion. This has earlier been proved by Reps and Yang [RY89]. They used deductive reasoning to establish their proofs. In contrast our proof procedures throughout the paper have calculational style. These proofs are "better" according Gries [Gri91], a proponent of this style because they are shorter and easy to internalize since they directly show the assertion being established by symbol manipulation.

Theorem 5.2 $G_P \not\triangleright s_1 \vec{\cup} G_P \not\triangleright s_2 = G_P \not\triangleright (s_1 \cup s_2)$.

Proof: Let $G_P = <V, E, D>$, $G_P \not\triangleright s_1 = <V_1, E_1, D_1>$, $G_P \not\triangleright s_2 = <V_2, E_2, D_2>$, and $G_P \not\triangleright (s_1 \cup s_2) = <V_{12}, E_{12}, D_{12}>$. From definition of graph slicing and the model of O-O program slicing above we know that: $V_{12} = V_1 \cup V_2$ and $E_{12} = E_1 \cup E_2$. We only have to prove that $D_{12} = D_1 \cup D_2$.

$D_1 \cup D_2$

$= (D \cap V_1^3) \cup (D \cap V_2^3)$

$\subseteq D \cap (V_1 \cup V_2)^3$

We now prove that $D \cap (V_1 \cup V_2)^3 \subseteq (D \cap V_1^3) \cup (D \cap V_2^3)$ i.e. if $(v_1, v_2, v_3) \in D_{12}$ then $(v_1, v_2, v_3) \in D_1 \cup D_2$.

$(v_1, v_2, v_3) \in D_{12}$

$= (v_1, v_2, v_3) \in D \cap (V_1 \cup V_2)^3$

$= (v_1, v_2, v_3) \in D \wedge (v_1, v_2, v_3) \in (V_1 \cup V_2)$

$\Rightarrow (v_1, v_2, v_3) \in D \wedge (v_3 \in V_1 \vee v_3 \in V_2)$

$\underline{v_3 \in V_1 \Rightarrow (\forall u.(u, v_3) \in E_1, u \in E_1)}$ and

$\underline{(v_1, v_2, v_3) \in D_{12} \Rightarrow (v_1, v_2, v_3) \in D \Rightarrow (v_1, v_3) \in E \wedge (v_2, v_3) \in E}$

$\Rightarrow (v_1, v_2, v_3) \in D \wedge \{v_3 \in V_1 \wedge (v_1, v_3) \in E_1 \wedge (v_2, v_3) \in E_1\} \vee \{v_3 \in V_2 \wedge (v_1, v_3) \in E_2 \wedge (v_2, v_3) \in E_2\}$

$\underline{E_1 = E \cap V_1 \times V_1, E_2 = E \cap V_2 \times V_2}$

$\Rightarrow (v_1, v_2, v_3) \in D \wedge (v_1, v_2, v_3 \in V_1 \vee v_1, v_2, v_3 \in V_2)$
$= (v_1, v_2, v_3) \in D \cap V_1^3 \vee (v_1, v_2, v_3) \in D \cap V_2^3$
$= (v_1, v_2, v_3) \in D_1 \vee (v_1, v_2, v_3) \in D_2 = (v_1, v_2, v_3) \in D_1 \cup D_2$. []

Modelling modified Weiser's slicing algorithm

Definition: An *abstract extended PDG* is a 7–tuple $<V, E, D, Var, Def, Use, Rd>$ where $<V, E, D>$ is an abstract PDG and $Var \cap V = \phi$ and $Rd \subseteq V \times V \times Var$, $Def \subseteq V \times Var$, $Use \subseteq V \times Var$, and $Var = (Def \cup Use) \uparrow_2$.

In the following discussion we denote the 7–tuple constructed above as R_P and $\rho(R_P) \triangleq <V, E, D>$.

The set Var denotes the set of variables in the program. The sets Def, Use, and Rd relate to the *def*, *use*, and *reaching definitions* of the program as follows:

$(s, x) \in Def$ iff $x \in def(s)$,
$(s, x) \in Use$ iff $x \in use(s)$, and
$(s_1, s_2, x) \in Rd$ iff $s_1 \in rd(s_2) \wedge x \in def(s_1)$.

Further, the set of variables in Var is the same as the set $(Def \cup Use) \uparrow_2$.

The definition of abstract extended PDG contains all the constraints involved in the construction of a PDG and the set of relations, except for the types associated to the vertices and edges of the PDG. To model the extended Weiser's slicing algorithm we define an auxilliary function as follows.

Definition: $\psi(R_P, N, X) \triangleq (N \times X \cap (Def \cup Use) \uparrow_1) \cup (V \times (N \times X - Def \cup Use) \cap Rd) \uparrow_1$. Whenever R_P can be determined unambiguously from the context the function is simply stated as $\psi(N, X)$.

This function is equivalent to the **if-then-else** expression in the definition of modified Weiser's slicing algorithm of Section 3.

Lemma 5.3 $s_1 \in \psi(\{s_2\}, \{x\})$ iff $((x \in use(s_2) \cup def(s_2)) \wedge s_1 = s_2) \vee (\neg(x \in use(s_2) \cup def(s_2)) \wedge s_1 \in rd(s_2) \wedge x \in def(s_1))$

Proof: Exercise.

Definition: $R_P \psi <s, x> \triangleq <V', E', D', Var', Def', Use', Rd'>$ such that:

$<V', E', D'> = \rho(R_P) \phi \psi(\{s\}, \{x\})$,
$Def' = Def \cap (V' \times Var)$,
$Use' = Use \cap (V' \times Var)$,
$Var' = (Def' \cup Use') \uparrow_2$, and
$Rd' = Rd \cap (V' \times V' \times Var')$.

Lemma 5.4 *$R_P \psi <s, x>$ as defined above is an abstract extended PDG for the program represented by the abstract PDG $\rho(R_P) \phi \psi(\{s\}, \{x\})$.*

Proof: Hint: *Feasibility Theorem of* [RY89] and construction. []

This implies that $\rho(R_P \psi <n, x>) = \rho(R_P) \phi \psi(\{n\}, \{x\})$.

We would now investigate how union of Weiser's slice relates to the union of their slicing criterion. To do so we first extend the definition of slice above to take a slicing criterion consisting of a 2–tuple consisting of a set of statements and a set of variables.[#]

Definition: The slice $R_P \not\psi <S, X>$ is defined by replacing $\psi(\{s\}, \{x\})$ by $\psi(S, X)$ in the previous definition.

Lemma 5.5 *If* $N_1 = N_2$ *or* $X_1 = X_2$ *then* $\psi(N_1, X_1) \cup \psi(N_2, X_2) = \psi(N_1 \cup N_2, X_1, X_2)$.

Proof: Exercise. *Hint: If* $N_1 = N_2$ *or* $X_1 = X_2$ *then* $N_1 \times X_1 \cup N_2 \times X_2 = (N_1 \cup N_2) \times (X_1 \cup X_2)$. []

Definition: An abstract extended PDG is *feasible* if there exists a program that corresponds to it; otherwise it is *infeasible*.

Theorem 5.6 $\exists R_P, N_1, X_1, N_2, X_2$ *such that* $R_P \not\psi <N_1, X_1> \vec{\cup}\ R_P \not\psi <N_2, X_2>$ *is not a feasible abstract extended PDG.*

Proof: *Hint:* A PDG is *adequate* to represent programs [HPR88b]. Thus the program represented by the union of abstract extended PDGs should be the same as that due to the union of the abstract PDGs contained in them. But the union of sets corresponding to the reaching definitions in $R_P \not\psi <N_1, X_1>$ $\vec{\cup}\ R_P \not\psi <N_2, X_2>$ may not be the same as the corresponding set in the abstract extended PDG for the program represented by $\rho(R_P \not\psi <N_1, X_1>) \vec{\cup}\ \rho(R_P \not\psi <N_2, X_2>)$. []

Due to the above result it is not worth investigating the union of abstract extended PDGs resulting from modified Weiser's slice. But from the point of view of program slicing the abstract extended PDG generated from slicing is not important, but the program represented by it is. This program is, as just stated, represented by the abstract PDG contained in the abstract extended PDG's. The following theorem then states the constraints under which a program slice generated by the union of Weiser's slicing criteria is the same as that due to the union of the independent slices.

Theorem 5.7 *If* $N_1 = N_2$ *or* $X_1 = X_2$ *then*

$$\rho(R_P \not\psi <N_1, X_1>) \vec{\cup}\ \rho(R_P \not\psi <N_2, X_2>) = \rho(R_P \not\psi <N_1 \cup N_2, X_1 \cup X_2>).$$

Proof: $\rho(R_P \not\psi <N_1, X_1>) \vec{\cup}\ \rho(R_P \not\psi <N_2, X_2>)$

$= \rho(R_P) \oint \psi(N_1, X_1) \vec{\cup}\ \rho(R_P) \oint \psi(N_2, X_2)$

Theorem 5.2

$= \rho(R_P) \oint (\psi(N_1, X_1) \cup \psi(N_2, X_2))$

$N_1 = N_2 \vee X_1 = X_2$, Lemma 5.5

$= \rho(R_P) \oint \psi(N_1 \cup N_2, X_1 \cup X_2)$

$= \rho(R_P \not\psi <N_1 \cup N_2, X_1 \cup X_2>)$ []

This result may be stated as: *For any program P if* $N_1 = N_2$ *or* $X_1 = X_2$ *then* $P \not\psi <N_1, X_1> \vec{\cup}$ $P \not\psi <N_2, X_2> = P \not\psi <N_1 \cup N_2, X_1 \cup X_2>$. The analogous assertion Wieser [Wei84] makes may be stated as: $P \not\psi <N, X_1> \vec{\cup}\ P \not\psi <N, X_2> = P \not\psi <N, X_1 \cup X_2>$. Our theorem therefore proves a more general assertion.

[#] Note that Weiser extended his slicing criterion only to allow set of variables and not set of statements. Our extension is thus more general than that of Weiser.

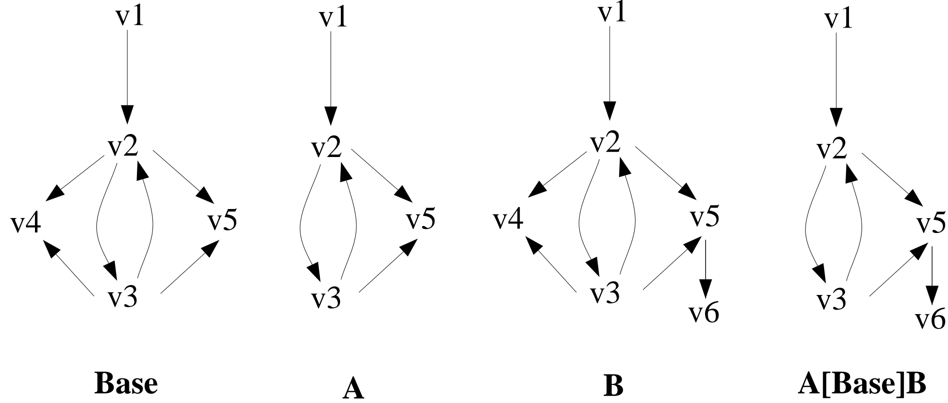

Figure 5 Example of *integrating* graph variants. **A** is created from **Base** by deleting vertex $v4$ and edges incident upon it.; **B** by adding a new vertex $v6$ and the edge $v5 \rightarrow v6$. These changes are preserved in **A[Base]B**.

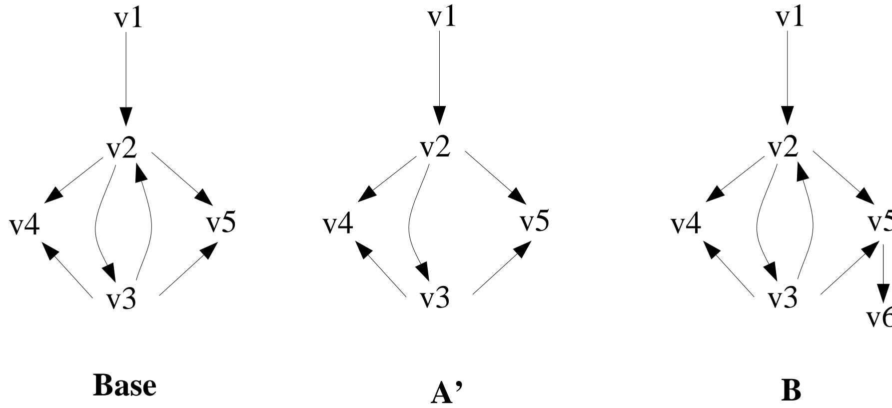

Figure 6 Example of interfering graph variants. **A'** is created by removing the edge $v3 \rightarrow v2$ from **Base**. **B** by adding the vertex $v6$ and the edge $v5 \rightarrow v6$. The two variants interfere and hence cannot be integrated.

6 Graph-theoretic foundations of integration

HPR Integration theorem

The previous section defined the notion of graph-slice as a parallel to program slice. We would now see what the HPR integration algorithm does if it operates on graphs (instead of PDGs) and uses graph slicing instead of program slicing. We call this *abstract HPR integration algorithm*.

Definition *(Abstract HPR integration algorithm):* Let a and b two graphs that are *variations* of $base$. Graph m, the integration of a and b is defined as follows:

$m = (a/ap_{a,base})\vec{\cup}(b/ap_{b,base})\vec{\cup}(base/pp_{base,a,b})$, where:

$$ap_{x,base} \triangleq \{v \in \nu(x) \mid (base/v) \neq (x/v)\} \text{ and}$$
$$pp_{base,x,y} \triangleq \{v \in \nu(base) \mid (base/v) = (x/v) = (y/v)\}.$$

The two graphs do not *interfere* if $m/ap_{a,base} = a/ap_{a,base}$ and $m/ap_{b,base} = b/ap_{b,base}$.

Note that the Type II interference (related to feasibility of PDG) can not be translated to directed graphs since there are no constraints on their structure.

In the rest of this section the symbols ap_a, ap_b, and pp are used as shorthand for $ap_{a,base}$, $ap_{b,base}$, and $pp_{base,a,b}$, respectively, as defined above. The symbols m, a, b, and $base$ have the same definitions as above.

Figure 5 gives an example of graph integration and Figure 6 of variations of graphs in which changes interfere. Their discussion is included in the caption itself.

Theorem 6.1 *(Structural integration theorem).* If graphs a and b do not 'interfere' with respect to $base$ then

1. $m/ap_a = a/ap_a$,
2. $m/ap_b = b/ap_b$, and
3. $m/pp = base/pp$.

Proof: The first two clauses follow from the definition of non-interference. The third clause is proved in Lemma 6.1.4. []

Definition: Henceforth the symbols pp_a and pp_b are used to define the following $pp_a \triangleq \nu(a/ap_a) - ap_a$ and $pp_b \triangleq \nu(b/ap_b) - ap_b$.

The following lemma defines some properties of the symbols defined above. They are stated without proof.

Lemma 6.1.1

1. $\nu(base/pp) = \nu(a/pp) = \nu(b/pp) = pp$.
2. $ap_a \cap pp = \phi, ap_b \cap pp = \phi$.
3. $a/pp_a = base/pp_a, b/pp_b = base/pp_b$.
4. $\nu(a/pp_a) = pp_a, \nu(b/pp_b) = pp_b$.
5. $pp = pp_a \cap pp_b$.
6. $pp_a - pp \subseteq ap_b, pp_b - pp \subseteq ap_a$.

Proof: Exercise. []

Lemma 6.1.2 $r \cap \nu(g/s) = \phi \equiv \epsilon(g) \cap r \times \nu(g/s) = \phi$.

Proof: Exercise. []

Lemma 6.1.3 $\nu(m) = ap_a \cup ap_b \cup pp$ and

$$\begin{aligned}\epsilon(m) = {} & \epsilon(a) \cap \{ap_a^2 \cup (pp_a \times ap_a) \cup pp_a^2\} \\ & \cup \epsilon(b) \cap \{ap_b^2 \cup (pp_b \times ap_b) \cup pp_b^2\} \\ & \cup \epsilon(base) \cap pp^2.\end{aligned}$$

Proof: $\nu(m)$

$$\begin{aligned}&= \nu((a/ap_a)\vec{\cup}(b/ap_b)\vec{\cup}(base/pp)) \\ &= \nu(a/ap_a) \cup \nu(b/ap_b) \cup \nu(base/pp) \\ &= (ap_a \cup pp_a) \cup (ap_b \cup pp_b) \cup pp \\ &\quad \underline{pp_a - pp \subseteq ap_b, pp_b - pp \subseteq ap_a, pp = pp_a \cap pp_b} \\ &= ap_a \cup ap_b \cup pp.\end{aligned}$$

$\epsilon(m)$
$= \epsilon((a/ap_a)\vec{\cup}(b/ap_b)\vec{\cup}(base/pp))$
$= \epsilon(a/ap_a) \cup \epsilon(b/ap_b) \cup \epsilon(base/pp)$
$= \epsilon(a) \cap (ap_a \cup pp_a)^2 \cup \epsilon(b) \cap (ap_b \cup pp_b)^2 \cup \epsilon(base) \cap pp^2$
$= \epsilon(a) \cap \{ap_a^2 \cup pp_a^2 \cup ap_a \times pp_a \cup pp_a \times ap_a\}$
$\cup \epsilon(b) \cap \{ap_b^2 \cup pp_b^2 \cup ap_b \times pp_b \cup pp_b \times ap_b\}$
$\cup \epsilon(base) \cap pp^2$
$\underline{(\nu(a/pp_a) = pp_a) \wedge (pp_a \cap ap_a = \phi), Lemma\ 6.1.2}$. Symmetrically for b.
$= \epsilon(a) \cap \{ap_a^2 \cup (pp_a \times ap_a) \cup pp_a^2\}$
$\cup \epsilon(b) \cap \{ap_b^2 \cup (pp_b \times ap_b) \cup pp_b^2\}$
$\cup \epsilon(base) \cap pp^2.$

Lemma 6.1.4 $m/pp = base/pp.$

Proof: From definition $pp \subseteq \nu(m/pp)$.

If $pp \neq \nu(m/pp)$ then $\exists k, k \neq \phi, k \cap pp = \phi$ such that $\nu(m/pp) = pp \cup k$ then from Lemma 6.1.2 $\epsilon(m) \cap k \times pp \neq \phi$. We will show that $\forall k, k \cap pp = \phi$, $\epsilon(m) \cap k \times pp = \phi$ implying thereby that $\nu(m/pp) = pp$.

$\epsilon(m) \cap \{k \times pp\}$
$= \epsilon(a) \cap \{ap_a^2 \cup (pp_a \times ap_a) \cup pp_a^2\} \cap \{k \times pp\}$
$\cup \epsilon(b) \cap \{ap_b^2 \cup (pp_b \times ap_b) \cup pp_b^2\} \cap \{k \times pp\}$
$\cup \epsilon(base) \cap pp^2 \cap \{k \times pp\}$
$= \epsilon(a) \cap \{(ap_a \cap k) \times (ap_a \cap pp) \cup (pp_a \cap k) \times (ap_a \cap pp) \cup (pp_a \cap k) \times (pp_a \cap pp)\}$
$\cup \epsilon(b) \cap \{(ap_b \cap k) \times (ap_b \cap pp) \cup (pp_b \cap k) \times (ap_b \cap pp) \cup (pp_b \cap k) \times (pp_b \cap pp)\}$
$\cup \epsilon(base) \cap (pp \cap k) \times (pp \cap pp)\}$
$\underline{pp \cap k = \phi, ap_a \cap pp = \phi, ap_b \cap pp = \phi, pp_a \cap pp_b = pp}$
$= \epsilon(a) \cap \{(pp_a \cap k) \times pp\}$
$\cup \epsilon(b) \cap \{(pp_b \cap k) \times pp\}$
$\underline{k \cap pp = \phi \Rightarrow pp_a \cap k \cap pp = \phi, \nu(a/pp) = pp \Rightarrow \forall r, r \cap pp = \phi, \epsilon(a) \cap r \times pp = \phi)}$ Symmetrically for b.
$= \phi$
$\Rightarrow \nu(m/pp) = \nu(base/pp) = pp.$

Now we show that $\epsilon(m/pp) = \epsilon(base) \cap pp^2 = \epsilon(base/pp)$.

$\epsilon(m/pp)$
$= \epsilon(a) \cap \{ap_a^2 \cup (pp_a \times ap_a) \cup pp_a^2\} \cap pp^2$
$\cup \epsilon(b) \cap \{ap_b^2 \cup (pp_b \times ap_b) \cup pp_b^2\} \cap pp^2$
$\cup \epsilon(base) \cap pp^2 \cap pp^2$
$= \epsilon(a) \cap \{(ap_a \cap pp)^2 \cup (pp_a \cap pp) \times (ap_a \cap pp) \cup (pp_a \cap pp)^2\}$
$\cup \epsilon(b) \cap \{(ap_b \cap pp)^2 \cup (pp_b \cap pp) \times (ap_b \cap pp) \cup (pp_b \cap pp)^2\}$
$\cup \epsilon(base) \cap pp^2\}$

$$\underline{ap_a \cap pp = \phi, ap_b \cap pp = \phi, pp_a \cap pp_b = pp}$$
$$= \epsilon(a) \cap pp^2 \cup \epsilon(b) \cap pp^2 \cup \epsilon(base) \cap pp^2$$
$$\underline{a/pp = b/pp = base/pp}$$
$$= \epsilon(base) \cap pp^2. \quad []$$

Reps' integration algorithm

Reps [Rep90] studies algebraic properties of program integration operation, such as whether there are laws of associativity and distributivity. For instance, associativity in the context of program integration means that "if three variants of a program are integrated by means of two-variant integrations, the same result is produced no matter which two variants are integrated first". To investigate such properties, Reps formulates the HPR integration algorithm as an operation in a *Brouwerian algebra* constructed from sets of dependence graphs. A Brouwerian algebra is a distributive lattice with an operation $a \dot{-} b$ characterized by $a \dot{-} b \sqsubseteq c$ iff $a \sqsubseteq b \sqcup c$. In this algebra he defines the program integration operation solely in terms of $\sqcup, \sqcap$, and $\dot{-}$ operations.

Just as for HPR integration algorithm, one can abstract Reps' integration algorithm for graph integration by simply replacing 'program' and 'program dependence graph' by 'directed graph' and 'slice' by 'graph slice' in Reps' description. This is to say that one can construct a Brouwerian algebra from sets of directed graphs as well.

Since the abstracted algorithm *reads* the same as the original algorithm, but for the use of different terms, we do not expand on it further. We leave it to the reader to validate our claims.

7 Semantics of graph slicing and integration

The graph slicing and integration algorithms developed abstracted various program slicing and integration algorithms without concern for any interpretation associated to the graph. Since we do neither require any constraints on the structure of the directed graphs nor on the contents of its vertex or edges we can not associate any interpretation to it that may be analogous to interpretations of programs.

One may however note that the graph integration operation is defined using graph slicing and other graph-theoretic operations. We therefore study the meaning of the result of integration in terms of *some abstract interpretation* and the slicing operation.

An interpretation, broadly speaking, is a mapping of elements of domain of study to a domain of interpretation.

Definition: Let $\mathcal{G}$ be the domain of graphs, $\mathcal{V}$ be the domain of vertices used in constructing elements of $\mathcal{G}$. Let $\mathcal{D}$ be the domain of interpretation with an *equality* function $\simeq$. Let $\pi : \mathcal{G} \times 2^{\mathcal{V}} \rightarrow \mathcal{D}$ be an *interpretation* of vertices in elements of $\mathcal{G}$. We denote $\pi_g(s)$ to be the interpretation of $g \in \mathcal{G}$ and $s \in 2^{\mathcal{V}}$.

We will show that if the interpretation π satisfies the following *slicing axioms* then the *semantic integration theorem*, stated ahead, is also be satisfied, irrespective of the specific interpretation and the structure of domain $\mathcal{D}$.

Axiom 7.1 *(Slicing axiom)* $\pi_g(s) \simeq \pi_{g/s}(s)$. []

Corollary: (Slicing corollary) $g/s = f/s \Rightarrow \pi_g(s) \simeq \pi_f(s)$. []

Definition: The interpretation of a graph g with respect to the graph f is said to be *similar* at vertex set x if the interpretation of g at x is the same as the interpretation of f at x i.e. $\pi_g(x) \simeq \pi_f(x)$; otherwise it is *changed* or *different*.

Definition: $SV_{f,g} \triangleq \{x \mid x \in \nu(g) \cap \nu(f) \wedge \pi_g(x) \simeq \pi_f(x)\}$, the set of vertices of graph g and f with similar interpretations.

Definition: $CV_{f,g} \triangleq \nu(f) - SV_{f,g}$, the set of vertices of graph f whose interpretations in f and g are different.

Theorem 7.2 *(Semantic Integration theorem)* Let m be the graph resulting from a successful integration of graphs a and b with respect to graph $Base$ and let π be an interpretation of these graphs. If π satisfies the slicing axiom then m preserves the:

changes in interpretation of a with respect to $base$, i.e.
$\pi_m(CV_{a,base}) \simeq \pi_a(CV_{a,base})$,
changes in interpretation of b with respect to $base$, i.e.
$\pi_m(CV_{b,base}) \simeq \pi_b(CV_{b,base})$,
similarity in interpretations of a, b, and $base$, i.e.
$\pi_m(sv) \simeq \pi_a(sv) \simeq \pi_b(sv)$, where $sv = SV_{a,base} \cap SV_{b,base}$.

Proof: Follows from Theorem 6.1 (the Structural Integration Theorem), slicing axiom, and the following observations.

$ap_a = CV_{a,base}$,
$ap_b = CV_{b,base}$, and
$pp = SV_{a,base} \cap SV_{b,base}$.

where ap_a, ap_b, and pp as defined in the previous section. []

8 Related Works

Venkatesh [Ven91] performs an analogous study of semantics of various types of program slices. He classifies the result of various slicing algorithms along three dimensions: static vs. dynamic, forward vs. backward, and closure vs. executable. The static and dynamic slices are as defined in Section 2.

A *forward* slices is the set of statements whose values would be affected by the values of a variable at the beginning of a program. A *backward* slice is the set of statements that affect the value of a variable at the end of a program. A *closure* slice is the set of statements related to a variable through a closure of some dependence. The set may neither form a syntactically valid program nor preserve the behavior of the original program at the statements in the slice. An *executable* on the other hand should be syntactically valid as well as preserve the behavior of the program at the statements enclosed in the slice.

In this paper the algorithms we model generate static backward executable slices. Forward slicing algorithms as in [Bin91] typically perform transitive closure of all outgoing edges in a graph. This can be modelled either by a) reversing the direction of the edges in a graph and performing graph slicing or b) by defining a dual of unreachable subgraph – a subgraph that cannot reach any vertex in its complement.

The dynamic slicing algorithms of Korel and Laski [KL88] and Agrawal and Horgan [AH90] are adaptations of static slicing algorithms of Weiser's [Wei84] and Ottenstein and Ottenstein's [OO84], respectively. Korel and Laski adapt Weiser's algorithm by performing incremental flow analysis on the 'trace' of a program's execution, where a trace is the sequence of statements representing the execution of the program for a given input. An analogous adaptation of our modified Weiser's algorithm may be performed to model their algorithm.

Agrawal and Horgan adapt O-O slicing algorithm by defining a notion of Reduced Dynamic Program Dependence Graph and the program trace. A dynamic slice of a program at some statement p for some input is the set of statements that reach p in the corresponding Reduced Dynamic Dependence Graph. Agrawal and Horgan's slicing method can therefore be modelled using our graph-theoretic framework.

The notion of closure and executable slice is related to the syntax and semantics of the program associated to a slice of a graph representation. The syntactic validity corresponds to the notion of *feasibility*- is there a program for which graph slice is a representation. This notion as well as that of semantics is outside the scope of the abstraction performed by a graph.

9 Conclusions

In this paper we develop the graph theoretic foundations of algorithms for program slicing and program integration. This is acheived by abstracting Ottenstein & Ottenstein's (O-O) [OO84] and Weiser's [Wei84] program slicing algorithms and Horwitz, Prins, & Reps' (HPR) [HPR88a] and Reps' [Rep90] program integration algorithms as operating on directed graphs instead of program dependence graphs. The benefits from the generalization are summarized below and elaborated upon later.

1. It provides a simpler platform to investigate algebraic properties such as intersection and union of slices.
2. It provides a uniform framework to model different program slicing algorithms and investigate properties of their slices without concern for the underlying representation or program semantics.
3. It classifies properties of program slices and program integration into structural and semantic category: the first of which may be proved by graph-theoretic reasoning itself.
4. It makes HPR and Reps' integration algorithms transparent of the underlying program representation and outlines the requirements for other representations to be qualified for use with these algorithms, see Figure 2.
5. The graph integration algorithm can be used to integrate versions of any artifact (not just program) that may be abstracted as graphs and the meaning of the resulting artifact can be derived from the meaning preserved by a slice in this representation, see Figure 2.

The first benefit can be seen from our discussions in Section 4 and 6. These sections, respectively, introduce the notions of graph slicing and graph integration without mentioning anything about programs or their representations. They present algorithms to perform the respective functions and establish properties of their results independent of any program of its representation.

Section 5 models O-O and Weiser's slicing algorithms using graph slicing. The O-O slicing algorithm is modelled using graph slicing and mathematical model of program dependence graph. Weiser's slicing algorithm is modelled in terms of O-O slicing algorithm and an extended PDG representation – PDG extended with three other program relationships.

This modelling has the benefit that properties of Weiser's slice and O-O slices can be compared mathematically. To do so we first generalize Weiser's slicing criterion – <statement, set_of_variables> to<set_of_statement, set_of_variables> and derive the following results:

1. The set of all O-O slices of a program forms a lattice with vector union, $\vec{\cup}$, as the meet operator and vector intersection, $\vec{\cap}$, as the join operator. The set of all Weiser's slice does *not* form a lattice with these operators.
2. We generalize Weiser's assertion that:
   $P/<l, x_1> \vec{\cup} P/<l, x_2> = P/<l, x_1 \cup x_2>$
   i.e. union of slices performed on the same statement over different variables is the same as the slice performed on that statements over the union of these variables,
   to: $P/<l_1, x_1> \vec{\cup} P/<l_2, x_2> = P/<l_1 \cup l_2, x_1 \cup x_2>$ if $l_1 = l_2$ or $x_1 = x_2$
   i.e. union of slices performed on a set of statements and a set of variable is the same as the slice performed on a union of these statements over the union of these variables, if either set of the statements are identical or set of the variables are identical.

Section 6 abstracts HPR integration algorithm over directed graphs. Since the algorithm is defined primarily using program slicing and set-theoretic operations its abstraction over directed graph is straightforward. Horwitz, Prins, and Reps have defined the meaning of the operands of integration and its result using the operational semantics of procedural programs.

We define the relationship between the operands and results of integration as two separate relationships. The first is a structural integration theorem which defines relationships between slices of the operands and results of integration over different sets of vertices representing changed and preserved behavior. The second is a semantic integration theorem which defines the relationship between 'meanings' of these programs using an *abstract interpretation*. It says that if the interpretation associated to a graph satisfies a *slicing axiom* (stated inside) then the result of integration will satisfy the *semantic graph integration theorem*.

This implies that HPR slicing and integration algorithms can be used with graph representations for programs other than PDGs for which the slicing axiom hold. A candidate for such a replacement is Yang's program representation graph [Yan90].

Restating HPR's integration theorem by abstracting the specific representation (PDG) and interpretation (operational semantics) has the benefit that this algorithm may now be used for integrating versions of any artifact that may be represented as graph. Further, for *every* interpretation associated to this graph that satisfies the slicing axiom one can derive the meaning of the result using the semantic integration theorem. Since graph notations such as data-flow diagrams, ER-diagrams, structure charts, state-transition diagrams are prevalent in representing specifications and designs [Pre87] it implies that one could *potentially* use the abstract HPR integration algorithm for integrating 'non-interfering' versions of specifications and designs as well. Its use however is subject to associating interpretations to these notations that would a) satisfy the slicing axiom and b) capture the intuitive or formal meaning we associate to these notations. Exploring these is beyond the scope of this paper.

Bibliography

[AH90] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. *ACM SIGPLAN Notices*, 25(6):246–256, June 1990.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Ber89] Claude Berge. *Hypergraphs: combinatorics of finite sets*. North-Holland, 1989.

[Bin91] David W. Binkley. *Multi-procedure Program Integration*. PhD thesis, University of Wisconsin Madison, WI, August 1991.

[Gal90] Keith B. Gallagher. *Using Program Slicing in Software Maintenance*. PhD thesis, University of Maryland, Baltimore, 1990.

[Gri91] David Gries. Teaching calculation and discrimination: A more effective curriculum. *Communications of the ACM*, 34(3):44–55, March 1991.

[Hec77] Matthew S. Hecht. *Flow Analysis of Computer Programs*. North-Holland, New York, 1977.

[HPR88a] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating non-interfering versions of programs. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego*, pages 133–145, 1988.

[HPR88b] Susan Horwitz, Jan Prins, and Thomas Reps. On the adequacy of program dependence graphs for representing programs. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego*, pages 147–158, 1988.

[HRB90] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, 1990.

[KL88] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29:155–163, 1988.

[KMC72] D. J. Kuck, Y. Muraoka, and S.C. Chen. On the number of operations simultaneously executable in FORTRAN-like programs and their resulting speed-up. *IEEE Transactions on Computers*, C-12(12), December 1972.

[OO84] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. *ACM SIGPLAN Notices*, 19(5), May 1984.

[OT89] Linda M. Ott and Jeffrey J. Thuss. The relationship between slices and module cohesion. In *Proceedings of the 12th International Conference on Software Engineering*, May 1989.

[Pre87] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw Hill, second edition, 1987.

[Rep90] Thomas Reps. Algebraic properties of program integration. In N. Jones, editor, *Lecture notes in Computer Science*, volume 432, pages 326–340. Springer-Verlag, New York, NY, 1990.

[RY89] Thomas Reps and Wuu Yang. The semantics of program slicing and program integration. In *Proceedings of the Colloquium on Current Issues in Programming Languages, (Barcelona, Spain, 1989), Lecture Notes in Computer Science*, volume 352, pages 360–374. Springer-Verlag, New York, NY, 1989.

[Ven91] G. A. Venkatesh. The semantic approach to program slicing. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 107–119, 1991.

[Wei84] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[Yan90] Wuu Yang. *A new algorithm for semantics-based program integration*. PhD thesis, University of Wisconsin Madison, WI, August 1990.