Precise Executable Interprocedural Slices

DAVID BINKLEY
Loyola College in Maryland

The notion of a *program slice*, originally introduced by Mark Weiser, is useful in program debugging, automatic parallelization, program integration, and software maintenance. A slice of a program is taken with respect to a program point p and a variable x; the slice consists of all statements of the program that might affect the value of x at point p. An interprocedural slice is a slice of an entire program, where the slice crosses the boundaries of procedure calls.

Weiser's original interprocedural slicing algorithm produces imprecise slices that are executable programs. A recent algorithm developed by Horwitz, Reps, and Binkley produces more precise (smaller) slices by more accurately identifying those statements that might affect the values of x at point p. These slices, however, are not executable. An extension to their algorithm that produces more precise executable interprocedural slices is described together with. a proof of correctness for the new algorithm.

CR Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs − *control structures, procedures, functions, and subroutines*; D.3.4 [**Programming Languages**]: Processors − *compilers, optimization*

General Terms: Algorithms, Design

Additional Key Words and Phrases: control dependence, data dependence, program dependence graph, program slicing

---

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. This concept, originally discussed by Mark Weiser in [17], can be used to isolate individual computation threads within a program. Slicing can help a programmer understand complicated code [17], can aid in debugging and software maintenance [7, 8, 12], and can be used for automatic parallelization [2, 16].

Horwitz, Reps, and Binkley identify two different but related "slicing problems" [9]. It is important to understand the distinction between them (the names come from [15]).

This work was supported in part by a summer research grant from Loyola College.

Author's address: Computer Science Department, Loyola College, 4501 North Charles Street, Baltimore Maryland 21210-2699. binkley@cs.loyola.edu

Version (1) (Closure Slice)

The slice of a program with respect to program point p and variable x consists of components (*e.g.*, statements and predicates) of the program that might affect the value of x at point p.

Version (2) (Executable Slice)

The slice of a program with respect to program point p and variable x consists of a reduced program that computes the same sequence of values for x at p. That is, at point p the behavior of the reduced program with respect to variable x is indistinguishable from that of the original program.

For in*tra*procedural slicing—slicing within a single procedure where the slice does not extend across procedure boundaries—a solution to Version (1) provides a solution to Version (2), since the "reduced program" required in Version (2) can be obtained by restricting the original program to just the components found in the solution for Version (1) [13].

For in*ter*procedural slicing—generating a slice of an entire program where the slice extends across the boundaries of procedure calls—restricting the original program to just the components found for Version (1) may yield a program that is syntactically incorrect (and thus certainly not a solution to Version (2)). The reason behind this phenomenon has to do with multiple calls to the same procedure: it is possible that the program elements found by an algorithm for Version (1) will include multiple calls on a procedure that have different subsets of the procedure's parameters.

Weiser's original interprocedural slicing algorithm provides a solution to Version (2) [17]; however, this algorithm potentially includes large portions of the program that do not affect the value for x at p because (a) it fails to account for the calling context of a procedure: if a slice includes one call-site on a procedure then the slice includes all call-sites on the procedure, and (b) call-sites are treated as indivisible components: if a slice includes one parameter, it must include all parameters.

Horwitz et. al. [9] compute the interprocedural slice on variables x at point p using the *system dependence graph*, which extends previous dependence representations to incorporate collections of procedures (with procedure calls) rather than just monolithic programs. Their algorithm, hereafter referred to as the HRB algorithm, identifies a subgraph of the system dependence graph whose components potentially affect the sequence of values for x at p. The HRB algorithm provides a solution to Version (1) that correctly accounts for procedure calling context and treats call statement as divisible components. Thus, it produces slices that are more precise (smaller) than those determined by Weiser's algorithm.[1] These slices, however, are not executable.

The algorithm discussed in this paper produces executable slices that are more *precise* than those produces by Weiser. Here the term "precise" means that the algorithm correctly handles calling context and that it may selectively choose parameters from call statements. Precise does not mean it produces the smallest possible slice, which would require solving unsolvable data-flow problems. As a consequence, the algorithm must make safe approximations: although it may include unnecessary statements, it does include all necessary statements.

---

[1] The HRB algorithm has a restriction not present in Weiser's algorithm: a slice can only be taken with respect to program point p and variable x if x is defined or used at p.

Executable slices are useful as they are required in a variety of applications. For example, the software maintenance technique discussed in [7] and algorithm for reducing the cost of regression testing presented in [6], both rely on being able to execute a slice. Executable slices can also be used to provide coarse grain parallelism from sequential programs by running separate slices on different processors. Finally, producing executable programs permits the execution behaviors of a program and its slices to be compared.

After providing a brief summary of the system dependence graph and the HRB slicing algorithm in Section 2, Section 3 first illustrates how a slice produced by the HRB algorithm may not correspond to a syntactically correct program. It then describes the steps necessary to extend the HRB algorithm to produce precise executable interprocedural slices. Section 4 proves that this new algorithm produces slices that satisfy in Version (2). Finally, Section 5 provides a brief summary.

2. THE HRB ALGORITHM

2.1. The System Dependence Graph

The HRB slicing algorithm makes use of the *system dependence graph*[2] (SDG) [9], which is summarized in this section. The SDG models a language with the following properties (the addition of input and output represents a slight extension to the definition given in [9]):

(1) A complete system consists of a single main procedure and a collection of auxiliary procedures.

(2) Parameters are passed by value-result.[3]

(3) Input and output are modeled as streams; thus, for example, the statement *print* (x) is treated as if it were the assignment statement "*output_stream* = *concatenate* (*output_stream*, x)."

The SDG for system S, denoted by G_S, contains a collection of *procedure dependence graphs* (PDGs), one for each procedure in the system, connected by interprocedural control- and flow-dependence edges. Procedure dependence graphs are similar to "program dependence graphs" [1, 10, 11].

The PDG for procedure P, denoted by G_P, contains vertices, which represent the components of P, and edges, which represent dependences between them. With the exception of call statements, a single vertex represents assignment statements, input statements and output statements, and the predicates of conditional (if) and while-loop statements. In addition, there is a distinguished vertex called the *entry vertex*, and an *initial-definition vertex* for each variable that may be used before being defined. Initial-definitions vertices represent the assignment of the value 0 to each variable that may be used before being defined.

A call statement is represented using a *call* vertex and four kinds of *parameter vertices* that represent parameter passing: on the calling side, parameter passing is represented by *actual-in* and *actual-out* vertices, which are control dependent (see below) on the call vertex; in the called procedure parameter passing

---

[2] Because the term "program dependence graph" is associated with graphs that represent single procedure programs, the term "system dependence graph" was chosen for the dependence graphs that represents multi-procedure programs [9]. Similarly, the term "system"—rather than "program"— is used to mean a program with multiple procedures.

[3] Techniques for handling parameter passing by reference and for dealing with aliasing are discussed in [9].

is represented by *formal-in* and *formal-out* vertices, which are control dependent on the procedure's entry vertex. Actual-in and formal-in vertices are included for every parameter and global variable that may be used or modified as a result of the call; formal-out and actual-out vertices are included for every parameter and global variable that may be modified as a result of the call. (Interprocedural data-flow analysis is used to determine which parameters and globals may be used and/or modified as a result of a procedure call [3, 4].)

Procedure dependence graphs include three kinds of intraprocedural edges: *control dependence edges*, *data dependence edges*, and *summary edges*. The source of a control dependence edge is either the entry vertex, a predicate vertex, or a call vertex. Each edge is labeled either **true** or **false**. A control dependence edge from vertex *v* to vertex *u* means that during execution, whenever the predicate represented by *v* is evaluated and its value matches the label on the edge to *u*, then the program component represented by *u* will eventually be executed provided the system terminates normally[4] (edges from entry and call vertices are always labeled **true**; these vertices are assumed to always evaluate to **true**).

A data dependence edge from vertex *v* to vertex *u* means that the system's behavior might change if the relative order of the components represented by *v* and *u* were reversed. There are two kinds of data dependence edges, *flow dependence edges* and *def-order dependence edges*: a flow dependence edge connects a vertex *v* that represents an assignment to a variable *x* to a vertex *u* that represents a use of *x* reached by that assignment. A def-order edge runs between two vertices, *v* and *u* that both represent assignments to variable *x* where both assignment reach a common use, and *v* lexically precedes *u* (*i.e.*, *v* is to the left of *u* in the program's abstract syntax tree).

The chief difficulty in interprocedural slicing is correctly accounting for the calling context of a called procedure. To address the calling-context problem, the system dependence graph includes *summary edges*: edges that represent *transitive* dependences due to the effects of procedure calls. A summary edge connects actual-in vertex *v* to actual-out vertex *u* if there is a path in the SDG from *v* to *u* that *respects calling context* by matching calls with returns.

Connecting procedure dependence graphs to form an SDG is straightforward, involving the addition of three kinds of interprocedural edges: (1) a *call* edge connects each call vertex to the corresponding procedure-entry vertex; (2) a *parameter-in* edge connects each actual-in vertex at a call-site to the corresponding formal-in vertex in the called procedure; (3) a *parameter-out* edge connects each formal-out vertex to the corresponding actual-out vertex at each call site on the procedure.

<< Insert Figure 1. >>

Example. Figure 1 shows an example system and the corresponding SDG.

---

[4] There are two ways in which a system may fail to terminate normally: (1) the system contains a nonterminating loop, or (2) a fault occurs, such as division by zero.

2.2. The HRB Slicing Algorithm

Interprocedural slicing can be defined as a reachability problem using the SDG, just as intraprocedural slicing is defined as a reachability problem using the program dependence graph [8]. The slices obtained using this approach are similar to those obtained using Weiser's interprocedural-slicing method [17]. However, this approach does not produce slices that are as precise as possible, because it considers paths in the graph that are not possible execution paths. For example, there is a path in the graph shown in Figure 1 from the vertex of procedure *Main* labeled "$a_{in} := sum$" to the vertex of *Main* labeled "$i := z_{out}$." However, this path corresponds to procedure *Add* being called by the first call in *Main*, but returning to the second call in *Main*, which is not possible. The value of *i* after the first call in *Main* is independent of the value of *sum* before the call, and so the vertex labeled "$a_{in} := sum$" should not be included in the slice with respect to the vertex labeled "$i := z_{out}$."

To achieve more precise interprocedural slices, an interprocedural slice of an SDG *G* with respect to a set of vertices *V* is computed using two passes over the graph. Summary edges are used in the algorithm to permit "moving across" a call site without having to descend into the called procedure; thus, there is no need to keep track of calling context explicitly to ensure that only legal execution paths are traversed. Both passes operate on the SDG, traversing edges to find the set of vertices that can reach a given set of vertices along certain kinds of edges. The traversal in Pass 1 starts from all vertices in *V* and goes backwards (from target to source) along flow edges, control edges, call edges, summary edges, and parameter-in edges, but *not* along def-order or parameter-out edges. The traversal in Pass 2 starts from all vertices reached in Pass 1 and goes backwards along flow edges, control edges, summary edges, and parameter-out edges, but *not* along def-order, call, or parameter-in edges. The result of an interprocedural slice consists of the sets of vertices encountered during Pass 1 and Pass 2. The HRB algorithm for finding the vertices of an interprocedural slice is stated in Figure 2.

<< Insert Figure 2. >>

Example. Figure 1 shows an HRB slice.

In the remainder of the paper, we use the operators *b1* and *b2* to designate the individual passes of the HRB slicing algorithm. In the terminology of Figure 2, they are defined as follows:

$$b1(G, V) \triangleq (\text{ReachingVertices}(G, V, \{\text{def-order, parameter-out}\}))$$
$$b2(G, V) \triangleq (\text{ReachingVertices}(G, V, \{\text{def-order, parameter-in, call}\})).$$

3. EXECUTABLE SLICES

This section describes the extension of the HRB algorithm that produces precise executable interprocedural slices. Before presenting this extension, we first consider an example of an HRB slice that is not executable. After this example, we discuss the three steps necessary to produce an executable system from an HRB slice. This section concludes with a discussion of the new slicing algorithm's complexity.

3.1. A Non-Executable Slice

For intraprocedural slicing, the slice of a dependence graph can be used to produce an executable program by restricting the original system to those elements whose vertices are in the slice of the dependence graph.

The same is not true for an HRB slice of an SDG. For example, the system produced by restricting system *S* shown in Figure 3 to statements whose vertices are in the slice also shown in Figure 3 is syntactically illegal because it contains a parameter mismatch: two call statements representing calls on a procedure contain different subsets of the procedure's parameters.

Making this system syntactically legal by simply adding parameter *c* to the second call statement on *P* leaves a semantically unsatisfactory system: it aborts with a division-by-zero error on the second call to *P*. Although a division-by-zero error could be caught by an exception handler, the important point here is that the system representing the slice computes values not computed by the original system. An example similar to the one in Figure 3 that replaced the division-by-zero error with an infinite loop would prevent the system that represents the slice from computing the same values as the original system at all points beyond the infinite loop. The presence of an infinite-loop cannot be caught by an exception handler or any other algorithmic means. In order to produce an executable slice, the components that take part in the computation of parameter's such as *c* must also be included in the slice.

<< Insert Figure 3. >>

When applied to the example shown in Figure 3, Weiser's interprocedural slicing algorithm produces an executable slice because it treats call statements as indivisible components. However, this causes unwanted program components to be included. For example, it causes *d* and consequently the computation of *d* to be included in the slice.

3.2. Executable Slices

Closer inspection of the HRB slice in Figure 3, reveals that there are two related mismatch problems:

(1) two call-sites on the same procedure may include different actual-in vertices, and
(2) two call-sites on the same procedure may include different actual-out vertices.

For example, the slice shown in Figure 3 contains actual-in vertices for both parameters at the first call-site on *P* and only one at the second call-site (these vertices are labeled "$x_{in} := a$" and "$y_{in} := b$", and $y_{in} := a$," respectively). The slice also has mismatched actual-out vertices: the first call-site has the actual-out vertex for the first parameter and the second call-site has the actual-out vertex for the second parameter.

This section describes the three steps used to produce an executable slice from an HRB slice. The first step adds vertices that remove actual-in vertex mismatches, the second step adds vertices that remove actual-out vertices, and the third step produces a system from the resulting set of vertices and the original system. Before describing the first step, we formalize the notion of a mismatch in the following definition.

DEFINITION. (Vertex Mismatch). A subset *V* of the vertices of SDG *G* contains an *actual-in vertex mismatch* if it contains a call-site on a procedure, the formal-in vertex for a parameter of that procedure, but not the actual-in vertex corresponding to the formal-in vertex at the call-site. Similarity, *V* contains an *actual-out vertex mismatch* if it contains a call-site on a procedure, the formal-out vertex for a parameter of that procedure, but not the actual-out vertex corresponding to the formal-out vertex at the call-site.

Note that the formal definition of an actual-in vertex mismatch differs from the intuitive definition given above. The formal definition implies the intuitive definition: when an HRB slice contains a formal-in vertex, it contains at least one corresponding actual-in vertex; consequently, if some other actual-in vertex that also corresponds to the formal-in vertex is not in the slice then two call-sites on the same procedure include different actual-in vertices. The same is true for actual-out vertices.

In the following discussion let V_{HRB} be the set of vertices in an HRB slice of system S. The first step of the algorithm removes actual-in vertex mismatches. If V_{HRB} has an actual-in mismatch because actual-in vertex v is not in V_{HRB} then, as illustrated by c in Figure 3, simply adding v to V_{HRB} is insufficient because an appropriate initial parameter value must be computed for the actual parameter represented by v.

In order to include the program components necessary to compute a safe value for such parameters, the vertices of the slice $b2(G_S, v)$ are added to V_{HRB}. A *b2* slice will include components from procedure P and procedures called by P, but not components of procedures that call P. The necessary components from these procedures are already in the slice unless there is a further actual-in mismatch. Because of this, the process of adding *b2* slices is repeated until no further actual-in vertex mismatches exist.

The choice of a *b2* slice over some other operation that would include call sites on P (*e.g.*, a *b1* slice or a full slice) avoids including unwanted call-sites on P. Both *b1* and full slices include all call-sites on a procedure, which may include unwanted call sites. In contrast, a *b2* slice, which includes none of a procedure's call-sites, combined with the definition of actual-in vertex mismatch, avoids including unwanted call-sites.

The second step in making an HRB slice executable removes actual-out vertex mismatches. Because missing actual-out vertices represent dead-code no additional slicing is necessary. Actual-out mismatches are removed by simply adding missing actual-out vertices to V_{HRB}.

The third step of the algorithm produces a system from the set of vertices in V_{HRB} and system S by projecting the statements of S that are represented in V_{HRB}. In other words, the statements of the sliced system are the statements of S represented by vertices in V_{HRB}; these statements appear in the same order and at the same nesting level as in S.

A complete algorithm for producing executable interprocedural slices is stated in Figure 4. For the example in Figure 3, the *b2* slice taken with respect to actual-in vertex "$x_{in} := c$" includes the statement "$c := 2$," which computes the necessary actual parameter value for c.

<< Insert Figure 4. >>

3.3. Complexity

The complexity of algorithm ExecutableSlice presented in Figure 4 is given in terms of the size of the input system S. The complexity of the HRB algorithm used in line 1 of ExecutableSlice is analyzed in [9]. It has two parts: the polynomial complexity of constructing the SDG for S and the linear complexity of slicing G_S. The construction cost is $O(TCS \times CS^2 + NP \times V^2)$, where TCS is the total number of call-sites in the system, CS is the maximum number of call sites in any one procedure, NP is the number of procedures in the system, and V is the maximum number of vertices (statements and predicates) in any one procedure. This assumes that the number of parameters associated with a procedure is bounded by a constant. The complexity of slicing G_S is linear in the size of the SDG. The size of the SDG is (conservatively) bounded by $O(NP \times V^2)$, since there are at most $O(V^2)$ edges in any one PDG.

The complexity of the extension is also linear. The complexity of all the *b2* slices that result from lines 2-3 is linear in the size of the SDG because no vertex currently in the slice need be considered. The complexity of adding vertices in line 5 and the complexity of projecting the final system *Slice* in line 6 are also linear. (This assumes there is a pointer from each vertex in the SDG to its originating line of source text.) In summary, the overall complexity of constructing an SDG is polynomial in the size of the system and the complexity of producing an executable interprocedural slice from the SDG is linear in the size of the SDG.

4. PROOF OF CORRECTNESS

Definitions

DEFINITION. (VerticesOfSlice). For system S and vertex set V, $VerticesOfSlice(S, V)$ refers to the vertices in V' at line [6] of Figure 4.

DEFINITION. (Induced Graph of G_S). For system S and set of vertices V, $InducedGraph(G_S, V)$ is the graph (V, E), where E is the set of edges from G_S whose end-points are in V. (For a def-order edge all three points involved must be in V).

DEFINITION. (*roll-out*). The *roll-out* of a system is produced by the exhaustive in-line expansion of call statements to produce a program without procedure calls. Each expansion step replaces a call statement with a new *scope statement*. Each scope statement, which creates a separate name space to avoid variable naming conflicts, contains the body of the called procedure and is parameterized by two lists of assignment statements, transfer-in statements and transfer-out statements, which make explicit the transfer of values into and out-of the name space for the scope. There is one transfer-in statement for every parameter or global variable represented by an actual-in vertex and one transfer-out statement for every parameter or global variable represented by an actual-out vertex.

In the presence of recursion, roll-out leads to an *infinite* program. (The meaning of an infinite program is defined by the least upper bound of the meanings of the finite programs that approximate it.) We use the notation *roll-out*(S) to denote the program produced by repeatedly expanding call statements in procedure *Main* of system S.

Previous Results

(1) THEOREM (SYNTACTIC CORRECTNESS THEOREM FOR INTRAPROCEDURAL SLICES [14]). *If Q is a slice of single-procedure program P taken with respect to vertex set V then G_Q is isomorphic*[5] *to* $InducedGraph(G_P, VerticesOfSlice(P, V))$.

---

[5] Two (system dependence) graphs G_1 and G_2 are isomorphic iff the following conditions are satisfied:

(1) There is a 1-to-1 mapping g from the vertex set of G_1 onto the vertex set of G_2 and for every v in G_1, v and $g(v)$ have the same text.

(2) There is a 1-to-1 mapping h from the edge set of G_1 onto the edge set of G_2 and for every edge e in G_1, e and $h(e)$ are of the same type (*e.g.*, both control edges) and have the same label.

(3) For every edge $v \longrightarrow u$ in G_1, $h(v \longrightarrow u) = g(v) \longrightarrow g(u)$.

When G_1 and G_2 are isomorphic or when we are trying to prove G_1 and G_2 are isomorphic, for brevity, we will say v and $g(v)$ are the *same* vertex and e and $h(e)$ are the *same* edge.

(2) THEOREM (SLICING THEOREM FOR SINGLE PROCEDURE PROGRAMS [14]). *For single-procedure program P, if G_Q is a slice (including induced edges) of G_P then for all program points in Q, Q and P produce the same sequence of values when evaluated on any input for which the evaluation of P terminates.*

COROLLARY. *The theorem applies to slices of infinite program roll-out(P).*

PROOF. *The technique discussed in Chapter 8 of [5] demonstrates that the theorem applies to a series of finite approximations to roll-out(P). Because the meaning of an infinite program is defined as the least upper bound of the meanings of its finite approximations, the theorem to extend to the infinite programs produced by roll-out.* □

(3) THEOREM (ROLL-OUT THEOREM [5]). *Roll-out is a semantics-preserving transformation: the sequences of values produced by the occurrences of s in roll-out(S) are the same as the sequences of values produced by s in each of its calling contexts in S.*

New Results

LEMMA (INDUCED GRAPH ISOMORPHISM LEMMA). *Let $Slice = ExecutableSlice(S, V)$. G_{Slice} is isomorphic to $InducedGraph\,(G_{S,}\ VerticesOfSlice\,(S, V))$.*

PROOF. There is a trivial transformation τ from a procedure with call statements to a procedure with sequences of assignment statements that have the same data-flow effect and hence the same flow dependence edges. (This transformation does not preserve the semantics of the procedure, just the dependence edges associated with calls by including assignment statements such as "$tmp := a$" for an actual parameter a that is used in the called procedure.) Because the Syntactic Correctness Theorem For Intraprocedural Slices implies there exists an isomorphism ϕ for the transformed versions of *Slice* and *S*, there is an isomorphism for the untransformed versions of *Slice* and *S*. This isomorphism is the composition $\tau \circ \phi \circ \tau^{-1}$. □

LEMMA. (SLICING LEMMA). *If $Slice = Executable\_Slice(S, V)$ then $G_{roll\text{-}out(Slice)}$ is a slice (including induced edges) of $G_{roll\text{-}out(S)}$.*

PROOF. It is sufficient to prove that

(1) $G_{roll\text{-}out(Slice)}$ is a subgraph of $G_{roll\text{-}out(S)}$, and

(2) For every vertex x^∞ in $VertexSet(G_{roll\text{-}out(Slice)})$ and every non-def-order edge $e^\infty = y^\infty \longrightarrow x^\infty$ in the edge set of graph $G_{roll\text{-}out(S)}$, edge e^∞ exists in the edge set of graph $G_{roll\text{-}out(Slice)}$.

(Point (2) ensures that the slice of $G_{roll\text{-}out(S)}$ never "escapes" to vertices not in $G_{roll\text{-}out(Slice)}$.)

Part (1) follows from the Induced Graph Isomorphism Lemma, which implies G_{Slice} is isomorphic to a subgraph of G_S and thus a subgraph of G_S. This relationship extends to $G_{roll\text{-}out(Slice)}$ and $G_{roll\text{-}out(S)}$: each vertex or edge occurrence in $G_{roll\text{-}out(Slice)}$ exists because of a vertex or edge in G_{Slice} and a sequence of call-sites that was expanded. Since this vertex or edge and the sequence of call sites are in G_S, the vertex or edge occurrence is in $G_{roll\text{-}out(S)}$. Edges from G_{Slice} (respectively G_S) have occurrences in $G_{roll\text{-}out(Slice)}$ ($G_{roll\text{-}out(S)}$) because each call statement expansion produces a copy of a procedure in which the same control and data dependences exists as in the original procedure.

To prove (2), assume x^∞, y^∞, and e^∞ are occurrences of x, y, and e, respectively. Vertex x is in G_{Slice}; vertex y and edge e are in G_S. The proof has two cases.

(a) y^∞ does not represent a transfer-in statement.

The Induced Graph Isomorphism Lemma implies that x is in *VerticesOfSlice* (S, V). Because y^∞ does not represent a transfer-in statement, e is an intraprocedural edge in G_S; thus, the *b2* slice of G_S that includes x in *VerticesOfSlice* (S, V) also includes y (this slice is taken in line [1] or [3] in Figure 4). Having x and y in *VerticesOfSlice* (S, V) and e in *InducedGraph* $(G_S$, *VerticesOfSlice* $(S, V))$ implies, by the Induced Graph Isomorphism Lemma, that y and e are in G_{Slice}. Finally, the expanding of the call-sites in *Slice* whose expansion leads to the creation of x^∞ produces an occurrence of y^∞; hence, x^∞, y^∞, and the edge connecting them e^∞ are in $G_{roll\text{-}out(Slice)}$.

(b) y^∞ represents a transfer-in statement.

If y^∞ represents a transfer-in vertex then x^∞ represents a statement in a called procedure and y represents an actual parameter before a call on this procedure. Unlike the previous case, a *b2* slice of x does not include y because a *b2* slice does not traverse parameter-in (or call) edges. However, the removal of actual-in vertex mismatches in lines [2]-[3] of Figure 4 ensures y is in *VerticesOfSlice* (S, V). The remainder of the proof is identical to the proof of Case (a). □

THEOREM (SLICING THEOREM.) *Let Slice = Executable_Slice(S, V). For each program component in V, Slice and S produce the same sequence of values when evaluated on any input for which S terminates.*

PROOF. First, since $G_{roll\text{-}out(Slice)}$ is a slice of $G_{roll\text{-}out(S)}$, the Corollary to the Slicing Theorem for Single Procedure Programs implies that, for each component of *roll-out*(*Slice*), *roll-out*(*Slice*) and *roll-out*(*S*) compute the same sequence of values and *roll-out*(*Slice*) terminates when *roll-out*(*S*) terminates. Because *roll-out* is a semantics preserving transformation these sequences are all computed by *Slice* and *S* and *Slice* terminates when *S* terminates. We must now show, for each component $c \in V$, that all calling contexts for c in *S* are in *Slice* and that these calling contexts are encountered in the same order during the execution of *Slice* and *S*. First, *b1*(S, c) (the first pass of an HRB slice) traverses call edges; therefore, *Slice* includes all calling contexts for c. Second, since the statements in *Slice* occur in the same relative order as in *S*, the calling contexts common to *Slice* and *S* are encountered in the same order during the execution of *Slice* and *S*. □

5. SUMMARY

This paper has presented the first algorithm for interprocedural slicing that correctly accounts for calling context and produces executable programs. This algorithm is an extension of the HRB interprocedural slicing, which “fills-in” parameter mismatches left by the HRB algorithm. This paper also proves that these slices capture a subcomputation of the original program.

ACKNOWLEDGEMENT

The comments of Art Delcher, the three reviewers, and the editor led to substantial improvements in the exposition of this paper. The suggestions of James Lyle and Keith Gallagher improved the examples.

REFERENCES

1. Allen, F., Burke, M., Charles, P., Cytron, R., and Ferrante, J. "An overview of the PTRAN analysis system for multiprocessing." Res. Rep. RC 13115, IBM T.J. Watson Research Center, Yorktown Heights, NY (September 1987).

2. Badger, L. and Weiser, M. "Minimizing communication for synchronizing parallel dataflow programs." in *Proceedings of the 1988 International Conference on Parallel Processing,* (St. Charles, IL, Aug. 15-19, 1988), Pennsylvania State University Press, University Park, PA(1988).

3. Banning, J.P. "An efficient way to find the side effects of procedure calls and the aliases of variables." in *Conference Record of the Sixth ACM Symposium on Principles of Programming Languages,* (San Antonio, TX, Jan. 29-31, 1979), ACM, New York, NY(1979), 29-41.

4. Barth, J.M. "A practical interprocedural data flow analysis algorithm." *Commun. of the ACM* **21**(9)(September 1978), 724-736.

5. Binkley, D. "Multi-procedure program integration." Ph.D. dissertation and Technical Report TR-1038, Computer Sciences Department, University of Wisconsin, Madison, WI (August 1991).

6. Binkley, D. "Using semantic differencing to reduce the cost of regression testing." in *Proceedings of the IEEE Conference on Software Maintenance,* (Orlando, FL, Nov, 1992), IEEE Computer Society, Washington, DC(1992), 41-50.

7. Gallagher, K.B. and Lyle, J.R. "Using program slicing in software maintenance." *IEEE Transactions on Software Engineering* **SE-17**(8)(1991), 751-761.

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

9. Horwitz, S., Reps, T., and Binkley, D. "Interprocedural slicing using dependence graphs." *ACM Trans. Program. Lang. Syst.* **12**(1)(January 1990), 26-60.

10. Kuck, D.J., Muraoka, Y., and Chen, S.C. "On the number of operations simultaneously executable in FORTRAN-like programs and their resulting speed-up." *IEEE Trans. on Computers* **C-21**(12)(December 1972), 1293-1310.

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

12. Lyle, J. and Weiser, M. "Experiments on slicing-based debugging tools." in *Proceedings of the First Conference on Empirical Studies of Programming,* (June 1986), Ablex Publishing Co.(1986).

13. Reps, T. and Yang, W. "The semantics of program slicing." TR-777, Computer Sciences Department, University of Wisconsin, Madison, WI (June 1988).

14. Reps, T. and Yang, W. "The semantics of program slicing and program integration." in *Proceedings of the Colloquium on Current Issues in Programming Languages,* (Barcelona, Spain, March 13-17,

1989), *Lecture Notes in Computer Science,* Vol. 352, Springer-Verlag, New York, NY(1989), 360-374.

15. Venkatesh, G.A. "The semantic approach to program slicing." *Proceedings of the ACM SIGPLAN 91 Conference on Programming Language Design and Implementation,* (Toronto, Ontario, June 26-28, 1991), *ACM SIGPLAN Notices*, (July 1991).

16. Weiser, M. "Reconstructing sequential behavior from parallel behavior projections." *Information Processing Letters* **17**(5)(October 1983), 129-135.

17. Weiser, M. "Program slicing." *IEEE Transactions on Software Engineering* **SE-10**(4)(July 1984), 352-357.

```
procedure Main
    sum := 0
    i := 1
    while i < 11 do
        call Add(sum, i)
        call Inc(i)
    od
    print (sum)
end
```

```
procedure Add(a, b)
    a := a + b
end
```

```
procedure Inc(z)
    call Add(z, 1)
end
```

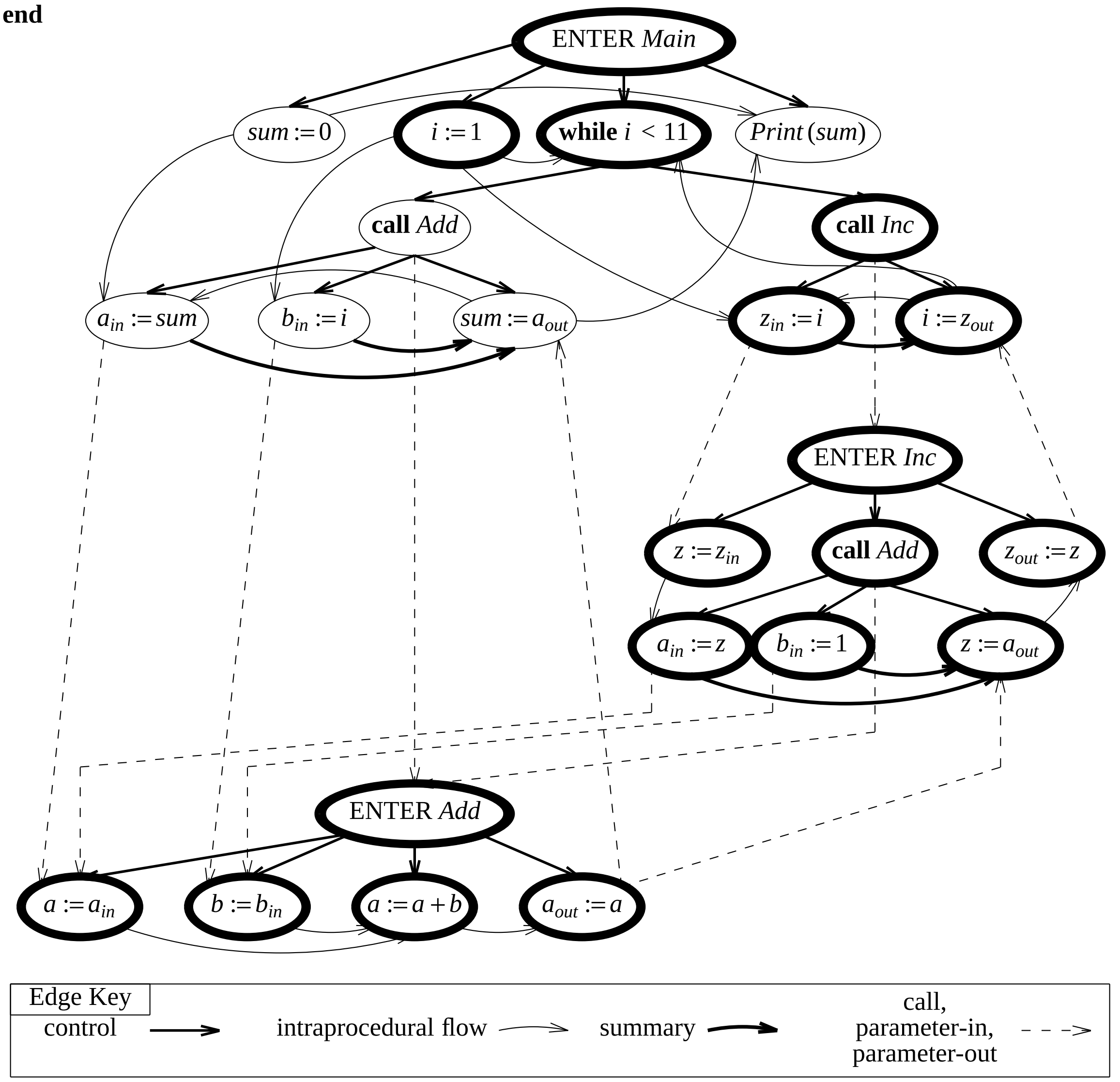


Figure 1. An example system, its SDG, and the slice taken with respect to the formal-out vertex labeled "$z_{out} := z$." The slice consists of the bold vertices. In comparison, Weiser's algorithm includes the entire system in the slice. (Def-order edges are not shown and the edges representing control dependences are shown unlabeled; all such edges would be labeled **true**.)

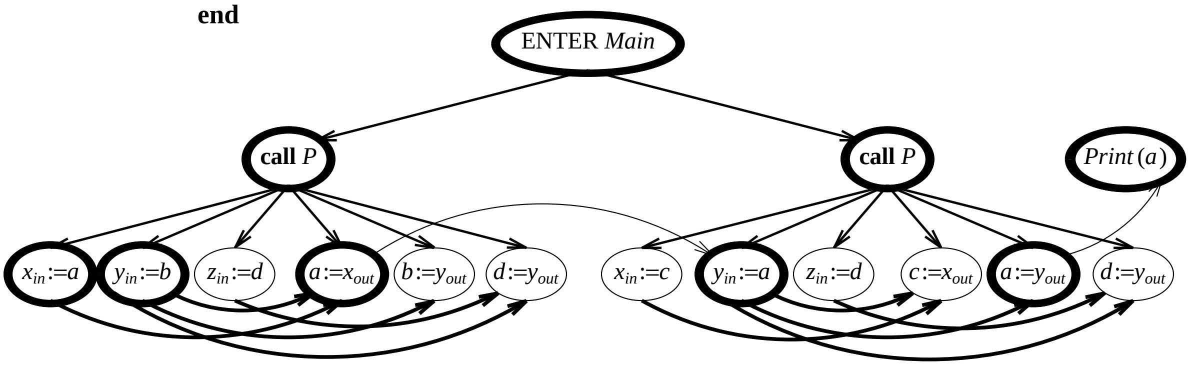
```
function VerticesOfSlice(G, V ) returns a set of vertices
declare
   G: an SDG
   V, V′, Answer: sets of vertices of G
begin
   Unmark all vertices of G
   /* Phase 1: Slice without descending into called procedures */
      V′ := ReachingVertices(G, V, {def-order, parameter-out})
   /* Phase 2: Slice called procedures without ascending to call sites */
      Answer := ReachingVertices(G, V′, {def-order, parameter-in, call})
   return(Answer)
end

function ReachingVertices(G, V, ExcludedEdgeKinds) returns a set of vertices
declare
   G: an SDG
   V, WorkList, Answer: sets of vertices of G
   ExcludedEdgeKinds: a set of edge-kinds
   v, w: vertices of G
begin
   WorkList := V
   Answer := ∅
   while WorkList ≠ ∅ do
      Select and remove a vertex v from WorkList
      Mark v
      Insert v into Answer
      for each unmarked vertex w such that there is an edge w → v whose kind is
                  not in ExcludedEdgeKinds do
         Insert w into WorkList
      od
   od
   return(Answer)
end
```

Figure 2. The function VerticesOfSlice returns the vertices of the interprocedural slice of SDG G with respect to vertex set V. The auxiliary function ReachingVertices returns all vertices in G from which there is a path to a vertex in V along edges whose edge-kind is something other than those in the set *ExcludedEdgeKinds*.

System S

procedure *Main*
    $c := 0$
    $a := 1$
    $b := c$
    $d :=$ some complex computation
    $c := 2$
    call $P\,(a,\ b,\ d)$
    call $P\,(c,\ a,\ d)$
    print (a)
end

procedure $P\,(x,\ y,\ z)$
    x := y / x
    y := y + 1
    z := z + y
end



System S restricted to elements whose vertices are in the above slice.

procedure *Main*
    $c := 0$
    $a := 1$
    $b := c$

    call $P\,(a,\ b)$
    call $P\,(\square,\ a)$
    print (a)
end

procedure $P\,(x,\ y)$
    x := y / x
    y := y + 1

end

Figure 3. The HRB slice of system S with respect to the vertex labeled “*print* (*a*)” (highlighted by the bold vertices) has mismatched actual-in and actual-out vertices. (Only part of the SDG for S is shown.)

```
     function ExecutableSlice(S, V) returns a system
     declare
        S: a system
        G_S: S's SDG
        Slice: the sliced system
        V, V′: sets of vertices of G_S
        x, y: vertices of G_S
     begin
[1]     V′ := HRB_slice(G_S, V)
[2]     while there exists an actual-in mismatch with formal-in vertex x in V′ and
                actual-in vertex y not in V′ do
[3]        Add the vertices in b2(G_S, y) to V′
        od

[4]     for all actual-out mismatches with formal-out vertex x in V′ and
                actual-out vertex y not in V′ do
[5]        Add y to V′
        od

[6]     Slice := Projection(S, V′)
[7]     return(Slice)
     end
```

Figure 4. The function ExecutableSlice returns a system that correctly captures the execution behavior of system S at each point in V.