Program Slicing

David W. Binkley
Keith Brian Gallagher

A hundred times every day, I remind myself that my inner and outer life depends on the labors of other men, living and dead, and that I must exert myself in order to give in the measure as I have received and am still receiving.

A. Einstein

1 Introduction

At the fifteenth International Conference on Software Engineering (ICSE), in 1993, Mark Weiser received recognition for the best paper at the fifth ICSE. Weiser reported that "Program Slicing"[94], the conference paper, and its journal incarnation of the same name[97], were cited about 10 times a year from 1984 to 1989 and about 20 times a year in 1990 and 1991. Our subsequent search of the "Science Citation Index" found 9 citations in 1992; 9 citations in 1993; and 15 citations in 1994. This is clearly a lower bound. Our bibliography of 94 items (91, excluding Weiser's original work) has only 12 that do not cite it, 5 of which were published prior to the original paper. The references included universities in 10 countries and 8 industrial laboratories. Since so many researchers and practitioners have been citing this seminal work, herein we attempt to collate, summarize and organize the edifice of results that have followed.

Program slicing is a decomposition technique that extracts from program statements relevant to a particular computation. Informally, a slice provides the answer to the question "What program statements potentially affect the computation of variable v at statement s?" An observer viewing the execution of a program and its slice cannot determine which is which when attention if focused on statement s. It is as if the observer has a window through which only part of the program state can be seen as illustrated by the "clear window" in Figure 1.

The utility and power of program slicing comes from the ability to assist in tedious and error prone tasks such as program debugging [1, 50, 54, 64, 65, 66, 88, 95], testing [3, 11, 17, 20, 31, 51, 54, 57, 87], parallelization [23, 96], integration [82, 43, 80, 81, 83], software safety [33], understanding [24, 56, 69, 75], software maintenance [32, 58, 70, 74, 77, 78], and

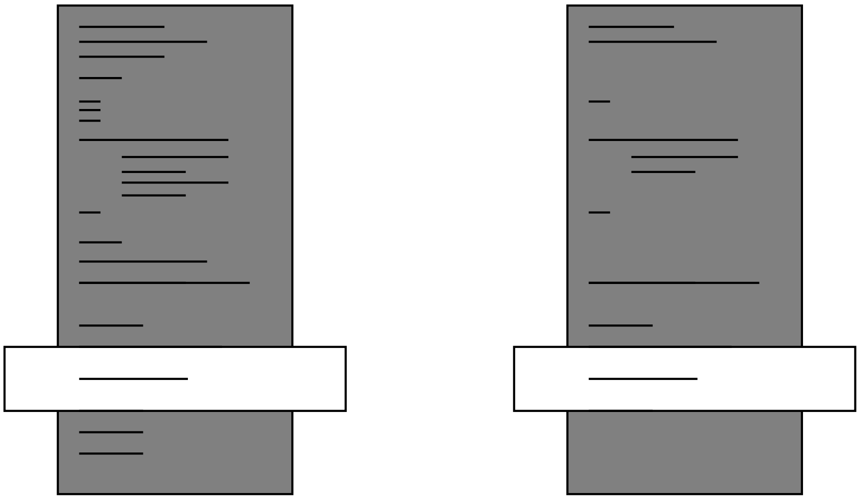

Figure 1: The Programmer's View of a Slice

software metrics [62, 63, 72, 71]. Slicing does this by extracting an algorithm whose computation may be scattered through out a program from intervening irrelevant statements. Consequently, it should be easier for a programmer interested in a subset of the program's behavior to understand the slice.

1.1 A Brief History

This section provides a brief history of program slicing and introduces the principal terminology. A slice is taken with respect to a *slicing criterion* $<s, v>$, which specifies a location (statement s) and a variable (v). This can be easily extended to slicing with respect to a collection of locations and a collection of variables at each location by taking the union of the individual slices. However, to simplify the exposition, we present definitions for the single statement and single variable case.

Program slices, as originally introduced by Weiser [93, 94, 97], are now called *executable backward static* slices. *Executable* because the slice is required to be an executable program. *Backwards* because of the direction edges are traversed when the slice is computed using a *dependence graph*. And finally, *static* because they are computed as the solution to a static analysis problem (*i.e.*, without considering the programs input).

Weiser's requirement that a slice be executable provided an empirical validation of the concept. Although this seems reasonable, many applications of program slicing (*e.g.*, debugging) do not require executable slices. This issue is considered in Section 2.

Weiser originally used a control-flow graph as an intermediate representation for his slicing algorithm. Ottenstein *et al.* [26, 73] later noted that backward slices could be efficiently computed using the program dependence graph as an intermediate representation by traversing the dependence edges backwards (from target to source). This observation was pursued vigorously and rigorously by Horwitz, *et al.* [40, 41, 42, 43, 44, 46, 47, 48, 82, 83, 86], who introduced the notion of *forward* slicing in [48]. Informally, a forward slice answers the question "What statements are affected by the value of variable v at statement s?"

Finally, Korel and Laski introduced the notion of *dynamic* slicing [53, 55]: a slice computed for a particular fixed input. The availability of run-time information makes dynamic slices smaller than static slices, but limits there applicability to that particular input. As with Weiser's algorithm, Korel and Laski's algorithm use a control-flow graph as an intermediate representation. Agrawal and Horgan later presented a dynamic slicing algorithm that used the program dependence graph as an intermediate representation [5, 6].

This chapter is organized into three major sections. Section 2 discusses static slicing as a data flow problem and as a graph reachability problem; then discusses dynamic slicing as a data flow problem and as a graph reachability problem; and closes with a collection of alternative methods for computing slices. The third section looks at applications of the idea of program slicing, without regard to any particular method of computing the slice.

2 Computing Slices

The following terminology is in this section to discuss the computation of slices.

Definition 1 Graph. A directed graph G is a set of nodes N and a set of edges $E \subseteq N \times N$. For edge $(n, m) \in E$, m is an *immediate successor* of n and n is an *immediate predecessor* of n. G contains two special nodes, $n_{initial}$, which has no predecessors, and n_{final}, which has no successors. Furthermore, there is a path from $n_{initial}$ to every node in G and a path from n_{final} to every node in G^{-1}, the inverse graph of G.

Definition 2 Control Flow Graph. A control flow graph for program P is a graph in which each node is associated with a statement from P and the edges represent the flow of control in P. Let V be the set of variables in P. With each node n (*i.e.*, each statement in the program and node in the graph) associate two sets: $REF(n)$, the set of variables whose values are referenced at n, and $DEF(n)$, the set of variables whose values are defined at n.

Definition 3 Program Slice. For statement s and variable v, the slice of program P with respect to the slicing criterion $< s, v >$ includes only those statements of P needed to *capture* the behavior of v at s.

Exactly what is meant be "capture" varies. The most intuitive definition is presented in the definition of an *executable slice*. Other definitions are examined later in this section.

n	statement
1	c = 0
2	while (TRUE)
3	c = 1
4	endwhile
5	c = 2

n	statement
5	c = 2

Figure 2: "Sliceable" Divergent Program and the convergent Slice taken with respect to $<5, c>$

Definition 4 Executable Program Slice. For statement s and variable v, the slice S of program P with respect to the slicing criterion $< s, v >$ is any executable program with the following properties:

1. S can be obtained by deleting zero or more statements from P.

2. If P halts on input I, then the value of v at statement n each time n is executed in P is the same in P and S. If P fails to terminate normally[1] n may execute more times in S that in P, but P and S compute the same values each time n is executed by P.

Note that every program has itself as a slice on any criteria.

EXAMPLE. The program of Figure 2 never executes Statement 5, while the slice on $<5, c>$ includes only Statement 5, which is executed once.

2.1 Slicing Control Flow Graphs

This section describes computing a slice as the solution to a data-flow problem using a control-flow graph as an intermediate representation. It considers a progression of harder slicing problems beginning with slicing straight-line programs, then considering structured control flow, unstructured control flow, data structures, and finally, procedures.

Computing a slice from a control-flow graph is a two step process: first requisite data flow information is computed and then this information is used to extract the slice. The data-flow information is the set of *relevant* variables at each node n. For the slice with respect to $< s, v >$, the relevant set for each node contains the variables whose values (transitively) affect the computation of v at s. The second step identifies the statements of the slice. These include all nodes (statements) n that assign to a variable relevant at n and the slice taken with respect to any predicate node that directly controls n's execution.

[1]A program fails to terminate normally if it diverges or terminates abnormally, for example, with a division by zero error.

n	statement	$refs(n)$	$defs(n)$	$relevant(n)$
1	$b = 1$		b	
2	$c = 2$		c	b
3	$d = 3$		d	b, c
4	$a = d$	d	a	b, c
5	$d = b + d$	b, d	d	b, c
6	$b = b + 1$	b	b	b, c
7	$a = b + c$	b, c	a	b, c
8	$print\ \ a$	a		a

Figure 3: Relevant Sets for $<8, a>$.

2.1.1 Slicing Flow Graphs of Straight Line Programs

Straight-line code contains only assignment statements executed one after the other. For such code the additional slices with respect to predicate nodes are not required. We begin by assuming that expression evaluation does not alter the values of its operands. *Relevant sets* for the slice taken with respect to $<n, v>$ are computed as follows [64]:

1. Initialize all relevant sets to the empty set.
2. Insert v into $relevant(n)$.
3. For m, n's immediate predecessor, assign $relevant(m)$ the value
   $$(relevant(n) - DEF(m)) \cup (REF(m) \text{ if } relevant(n) \cap DEF(m) \neq \emptyset)$$
4. Working backwards, repeat Step (3) for m's immediate predecessors until $n_{initial}$ is reached.

EXAMPLE. Figure 3 shows the relevant sets for a slice taken with respect to $<8, a>$. The relevant sets are computed from Line 8 to Line 1. For example, $relevant(7) = (relevant(8) - DEF(7)) \cup (REF(7) \text{ if } relevant(8) \cap DEF(7) \neq \emptyset) = (\{a\} - \{a\}) \cup (\{b, c\} \text{ if } \{a\} \cap \{a\} \neq \emptyset) = \{b, c\}$, and $relevant(2) = (\{b, c\} - \{c\}) \cup (\emptyset \text{ if } \{b, c\} \cap \{c\} \neq \emptyset) = \{b\}$. If we were interested in some variable other than a, the computation of the relevant sets would be different; thus, different relevant sets must be computed for different slices.

The relevant sets may be viewed as a flow of sets of variables; the slice is the set of statements that disturb this flow. If no relevant variables are defined at a statement, then the relevant set flows through unperturbed. On the other hand, if a relevant variable is defined then the statement is added to the slice. In Figure 3, for example, the slice with respect to $< 8, a >$ includes Lines 7, 6, 2 and 1 (Line 6 changes the b that is in the relevant set). Note that, in practice, the relevant sets and the statements in the slice are computed in a single pass.

n	statement	$refs(n)$	$defs(n)$	$control(n)$
1	$b = 1$		b	
2	$c = 2$		c	
3	$d = 3$		d	
4	$a = d$	d	a	
5	$if\ (a)\ then$	a		
6	$\quad d = b + d$	b, d	d	5
7	$\quad c = b + d$	b, d	c	5
8	$else$			5
9	$\quad b = b + 1$	b	b	8
10	$\quad d = b + 1$	b	d	8
11	$endif$			
12	$a = b + c$	b, c	a	
13	$print\ \ a$	a		

Figure 4: Control sets.

2.1.2 Slicing Structured Programs

In straight-line programs each statement has a single unique predecessor. In the presence of structured control flow, a statement may have multiple control predecessors. The above algorithm requires three modifications to handle this: first the inclusion of control sets, second a rule for combining relevant sets at control join points, and finally, iteration of the relevant set computation.

The control set $control(n)$ associates with node (statement) n, the set of predicate statements that directly control n's execution [64]. The use of a set here facilitates the transition to unstructured control-flow in Section 2.1.3. For a structured program, $control(n)$ contains a single entry, the loop or conditional statement that controls the execution of n, or is empty, when n is a *"top-level"* statement. For example, in Figure 4 $control(6)$ includes Statement 5, the $if\ (a)\ then$; the control sets for Statement 9 includes the Statement 8, the $else$, whose control set also includes Statement 5. Whenever a statement is added to the slice, the members of its control set, k, are added to the slice along with statements in the slice taken with respect to $<k, REF(k)>$.

At join points (where two nodes have the same predecessor), the relevant set is the union of the relevant sets of the two nodes. For example, in Figure 5, $relevant(5)$ is the union of $relevant(6)$ and $relevant(9)$. This assumes the expression in the conditional at the join point has no side effects. If it does then after the union, Step (3) of the relevant set computation must be performed to update the set to account for the side-effects of the expression.

Example. Figure 5 shows the data necessary to calculate the slice with respect to $<13, a>$ of the program shown in Figure 4. The slice is normally computed in conjunction with the

n	$refs(n)$	$refs(n)$	$control(n)$	$relevant(n)$
1		b		a
2		c		a, b
3		d		a, b
4	d	a		a, b
5	a			b, c, d
6	b, d	d	5	b, d
7	b, d	c	5	b, c
8			5	c, d
9	b	b	8	c, d
10	b	d	8	c, d
11				b, c
12	b, c	a		b, c
13	a			a

Figure 5: Relevant Sets on a at 13

n	statement
0	*while* (a)
1	$x_n = x_{n-1}$
2	$x_{n-1} = x_{n-2}$
	...
n	$x_2 = x_1$
n + 1	*endwhile*

Figure 6: A while statement that needs n passesto compute a slice when x_1 is in the criteria.

relevant sets: Working backwards from Line 13, since $DEF(12) \bigcap Relevant(13) \neq \emptyset$, Line 12 is included in the slice and its relevant set is assigned {b, c}. No change occurs at Line 11. Line 10 is included in the slice because $DEF(10) \bigcap Relevant(11) \neq \emptyset$; $relevant(10)$ is assigned {c, d}. Next, Lines 5 and 8 are included in the slice because $control(10)$ includes 8 and $control(8)$ includes Line 5. Along with these lines, the lines in the slices with respect to $<8, REF(8)> = <8, \emptyset>$ and $<5, REF(5)> = <5, a>$ are also included. These add to the slice Lines 4 and 3. Finally, Line 6 completes the slice.

The third change is required to handle loops. The absence of loops allows a slice to be computed in a single pass over the control-flow graph; the presence of loops requires iterating over parts of the graph. In particular, iteration over each loop until the relevant sets and slice stabilize. Hausler [39] has shown that the maximum number of iterations is the same as the number of assignment statements in the loop. Figure 6 shows how the upper bound is reached.

2.1.3 Slicing Un-Structured Programs

The addition of *goto* statements, and its restricted forms such as *return*, *exit*, *break*, and *continue* complicate the construction of the control sets. One solution is to restructure the program as *goto*-less [60] and then slice. The drawback to this method is that the structured version may be significantly textually dissimilar from the original.

Lyle [64] proposed a simple and conservative solution to slicing over unstructured control flows: if a *goto* statement has a non-empty relevant set, include it the slice. The targets of the *goto* are included, and so on until the slice stabilizes.

An alternative approach [29] is to note that *goto* statements are associated with *labels*. Rather than look for *goto* statements to include in the slice, look for labeled statements that are included in the slice. Then include only the *goto* statements that branch to these labels. This algorithm was devised for a language in which the only labels are targets of *goto*'s, a labeled statement does no computation, and does not have explicit compound statements. That is, statements of the form

```
if ( b ) then {
    /* compute X */
} else {
 /* compute Y */
}
```

are replaced by the semantically equivalent:

```
if ( b )  goto L1
      /* compute Y */
goto L2
L1:
      /* compute X */
L2:
```

Each statement of the form

```
label :  statement ;
```

is transformed to the semantically equivalent

```
label :  null ;
statement ;
```

Definition 5 Labeled block. *A* labeled block *is a basic block* [2] *that begins with a labeled statement and ends when one of the following occurs.*

1. *The basic block ends.*
2. *Another* labeled block *begins; i.e., another labeled statement occurs.*

Definition 6 Pseudo-label. *Let* L *be the label of the statement that begins a labeled block* B*. The remaining statements in* B *are said to have* pseudo-label L.

On the pass after a pseudo-labeled statement is added to the slice, as each *goto* statement is examined it is placed in the slice according to whether or not its target (pseudo-)label has already been added to the slice. The non-computing labeled statements are added if the actual label matches a pseudo-label in the slice.

2.1.4 Arrays, Records, and Pointers

To handle composite structures and pointers requires a change to the $DEF(n)$ and $REF(n)$ sets. A simple approach for arrays is to treat each array assignment as both an assignment and a use of the array. For

n: $a[i] := x;$

[2] A *basic block*[52] is sequence of consecutive instructions that are always executed from start to finish.

one may naively assume that $DEF(n) = \{\mathrm{a}\}$ and $REF(n) = \{i, x\}$. But the new value of a also depends and the old value of a, so a must also be included. The correct value for $REF(n)$ is $\{a, i, x\}$. However, this approach leads to correct, but unnecessary large slices.

To more precisely determine if there is a dependence between a statement that contains an assignment to $a[f(i)]$ and a statement that contains a use of $a[g(j)]$ requires answering the question "Can $f(i) = g(j)$?" In general this question is unanswerable, although it can be solved for some common index expression types. These solutions are often *one sided*: algorithms exist to determine if the answer to the question is "no." Otherwise, no information is obtained. To illustrate this, consider the Greatest Common Divisor (GCD) Test applied to the following loop

```
i = 0
while (i < N)
{
    X[a1 * i + a0 ] = ...
    ...= X[b1 * i + b0 ]
    i = i + 1
}
```

If $gcd(a_1, b_1)$ does not divide $(b_0 - a_0)$ then the GCD test demonstrates the absence of a dependence between s_1 and s_2, but if $gcd(a_1, b_1)$ divides $(b_0 - a_0)$, the solution may lay outside the range $0...N$. Other tests [68, 76] are similarly one sided. If none of these tests can prove the absence of a flow dependence then one is assumed to exist. This provides a more precise, yet safe, approximation to the correct data relationships. Once this has been done, the slicing algorithm proceeds as in the absence of arrays.

Records are simple, if not tedious, to treat in a slicing algorithm. Unlike arrays, each field of a record is identified by a constant (the field's name). This allows occurrences of $record.field$ to be treated as occurrences of the simple variable $record\_field$. The assignment of one record to another is modeled as a sequence of field assignment.

The multiple levels of indirection for both assignment and reference of pointers create difficult problems. One must obtain every possible location to which a pointer could point. If a variable is *def*ined or *ref*erenced through a chain of pointers (e.g., *****a), then all intermediate locations in the accessing of the variable must be considered as *ref*ed locations.

Lyle, et al. [67] construct, then prune a *pointer state graph* (PSS) for expression $*^k a$. The $PSS_k(a)$ is a directed graph with single source a and defined recursively as

$$PSS_k(a) = \begin{cases} a, & if\ k = 0 \\ \{v \mid *a = v\}, & if k = 1 \quad \& \ edge(a, v) \in PSS_k(a) \\ \{v \mid v \in PSS_1(u) \wedge u \in PSS_{i-1}(a)\}, & otherwise \quad \& \ edge(u, v) \in PSS_k(a) \end{cases}$$

$PSS_k(a)$ gives the possible references of $*^k a$. It is then pruned: $R_{PSS_k(a)}(w) = \{v \in PSS_k(a) \wedge dist(v, w) \leq k\}$, where $dist(w, v)$ is the distance, measured in edges, between v and w. $R_{PSS_k(a)}(w)$

is used to compute $DEF(n)$ by eliminating the indirect definitions of w that cannot reached via k levels of indirection from a. See [67] for a detailed discussion.

2.1.5 Interprocedural Slicing

Slicing across procedures[3] complicates the situation due to the necessity of translating and passing the criteria into and out of calling and called procedures. When procedure P calls procedure Q at statement i, the active criteria must first be translated into the context of Q and then recovered once Q has been sliced.

To translate a set of criteria, C, into a called procedure, for each $v \in relevant(succ(Q)) \cap actual\_parameter(Q)$ map $v \rightarrow \omega$, the corresponding formal parameter of Q. Then generate new criteria $< n^Q_{final}, \omega >$. If $v \in relevant(succ(Q)) \cap local\_definition(Q)$, i.e., a local redefinition of v occurs, change the line number of criteria involving v to i, the call-site of Q. When $n^Q_{initial}$ is reached, unmap $\omega \rightarrow v$, and replace $n^Q_{initial}$ with i. Weiser called this new set of criteria $DOWN_0(C)$.

This is essentially an in-line replacement of each procedure occurrence, with appropriate substitutions. Globals pass into the called procedure undisturbed if their visibility is not blocked by a variable in Q; thus references to them inside the procedures are captured correctly. This method was introduced early in the development of the idea of slicing and does not address hard questions about pointers, aliasing, function parameters, etc.

When P is called from statement j of Q, criteria must be generated to slice up into Q. The new criteria are generated in similar fashion to the calling context. Criteria involving local variables are discarded, so that no undefined references are passed out. Criteria involving formal parameters are mapped into the corresponding actual of Q with new line number j. Weiser called this set $UP_0(C)$.

The sets $DOWN_0(C)$ and $UP_0(C)$ are then mapped to functions from criteria to criteria: $DOWN(CC) = \cup_{C \in CC} DOWN_0(C)$ and $UP(CC) = \cup_{C \in CC} UP_0(C)$. Union and transitive closure are defined in the usual way for these relations. Thus $(DOWN \cup UP)^*(C)$ will give a complete set of criteria to obtain the interprocedural slice for any criteria. This conservative, but correct, approximation was improved by [48].

2.2 Slicing as a Graph-Reachability Problem

Ottenstein and Ottenstein [73] observed that the program dependence graph (PDG), used in vectorizing and parallelizing compilers and program development environments, would be an ideal representation for constructing program slices: "Explicit data and control dependence make the PDG ideal for constructing program slices" [73].

[3] This section follows closely that of Weiser [97].

This section first discusses a variation of the PDG used to compute intraprocedural slices of structured programs and the extensions necessary to slice programs that contain procedures and procedure calls[43]. Both of these algorithm have three steps:

1. Construct a dependence graph from the program.
2. Slice the dependence graph.
3. Obtain a sliced program from the sliced graph.

The advantage of the dependence graph approach is that steps 2 and 3 are efficient. For example, Step 2 is linear in the size of the graph. However, step 1, the construction of the pdg is, $O(n^2)$ in the number of statements in the program. In order to focus on the slicing algorithms, this section does not discuss how Step 1, dependence graph construction, is accomplished (see[26, 38, 43, 48, 61, 73] for details).

2.2.1 Intraprocedure Slicing

The PDG used for intraprocedural slicing by Horwitz, et al. [43] is a modified version of the dependence graph considered by Ottenstein and Ottenstein [73]. The PDG for program P, denoted by G_P, is a directed graph whose vertices are connected by several kinds of edges. The vertices of G_P represent the assignment statements and control predicates that occur in program P. In addition, G_P includes a distinquished vertex called the *entry vertex*.

G_P is a multi-graph. Not only can it have more than one edge between two vertices, it may have more than one edge of a given kind between two vertices. Edges in G_P represent *dependences* among program components. An edge represents either a *control dependence* or a *flow dependence*. Control dependence edges are labeled **true** or **false**, and the source of a control dependence edge is always the entry vertex or a predicate vertex. A control dependence edge from vertex u to vertex v, denoted by $u \to_c v$, means that during execution, whenever the predicate represented by u is evaluated and its value matches the label on the edge to v, then the program component represented by v will eventually be executed if the program terminates. For structured languages control dependences reflect the program's nesting structure. Consequently, PDG G_P contains a *control dependence edge* from vertex u to vertex v of G_P iff one of the following holds:

1. u is the entry vertex, and v represents a component of P that is not nested within any loop or conditional; these edges are labeled **true**.

2. u represents a control predicate, and v represents a component of P immediately nested within the loop or conditional whose predicate is represented by u.

If u is the predicate of a while-loop, the edge $u \to_c v$ is labeled **true**; if u is the predicate of a conditional statement, the edge $u \to_c v$ is labeled **true** or **false** according to whether v occurs in the **then** branch or the **else** branch, respectively.

A flow dependence edge from vertex u to vertex v means that the program's computation might be changed if the relative order of the components represented by u and v were reversed. The flow-dependence edges of a PDG are computed using data-flow analysis. A PDG contains a flow dependence edge from vertex u to vertex v, denoted by $u \rightarrow_f v$, iff all of the following hold:

1. u is a vertex that defines variable x.
2. v is a vertex that uses x.
3. Control can reach v after u via an execution path along which there is no intervening definition of x.

Flow dependences can be further classified as *loop carried* or *loop independent*. A flow dependence $u \rightarrow_f v$ is carried by loop L, denoted by $u \rightarrow_{lc(L)} v$, if in addition to 1, 2, 3 above, the following also hold:

4. There is an execution path that both satisfies the conditions of (3) above and includes a backedge to the predicate of loop L.
5. Both u and v are enclosed in loop L.

A flow dependence $u \rightarrow_f v$ is loop independent, denoted by $u \rightarrow_{li} v$, if in addition to 1, 2, 3 above, there is an execution path that satisfies 3 above and includes *no* backedge to the predicate of a loop that encloses both u and v. It is possible to have both $u \rightarrow_{lc(L)} v$ and $u \rightarrow_{li} v$.

When there is more than one loop-carried flow dependence edge between two vertices, each is labeled by a different loop that carries the dependence. Figure 7 shows an example program and its PDG.

For vertex s of PDG G, the *slice* of G with respect to s, denoted by $Slice(G, s)$, is a graph containing all vertices on which s has a transitive flow or control dependence (*i.e.*, all vertices that can reach s via flow and/or control edges):

$V(Slice(G, s)) = \{v \in V(G) \mid v \rightarrow^*_{c,f} s\}$.

The edges of $Slice(G, s)$ are those in the subgraph of G induced by $V(Slice(G, s))$:

$$\begin{aligned} E(Slice(G, S)) &= \{v \rightarrow_f u \in E(G) \mid v, u \in V(Slice(G, S))\} \\ &\quad \cup \{v \rightarrow_c u \in E(G) \mid v, u \in V(Slice(G, S))\} \end{aligned}$$

The vertices of the slice of the PDG shown in Figure 7 taken with respect to the output vertex for i are highlighted in Figure 7.

Slicing programs with composite data structures involves changing the computation of the flow dependence edge only. Two methods for slicing in the presence of arbitrary control flow (programs containing *gotos*) require modifying the control dependence subgraph of the PDG, but not the slicing algorithm. Choi and Ferrante [22] augment the the control flow graph that

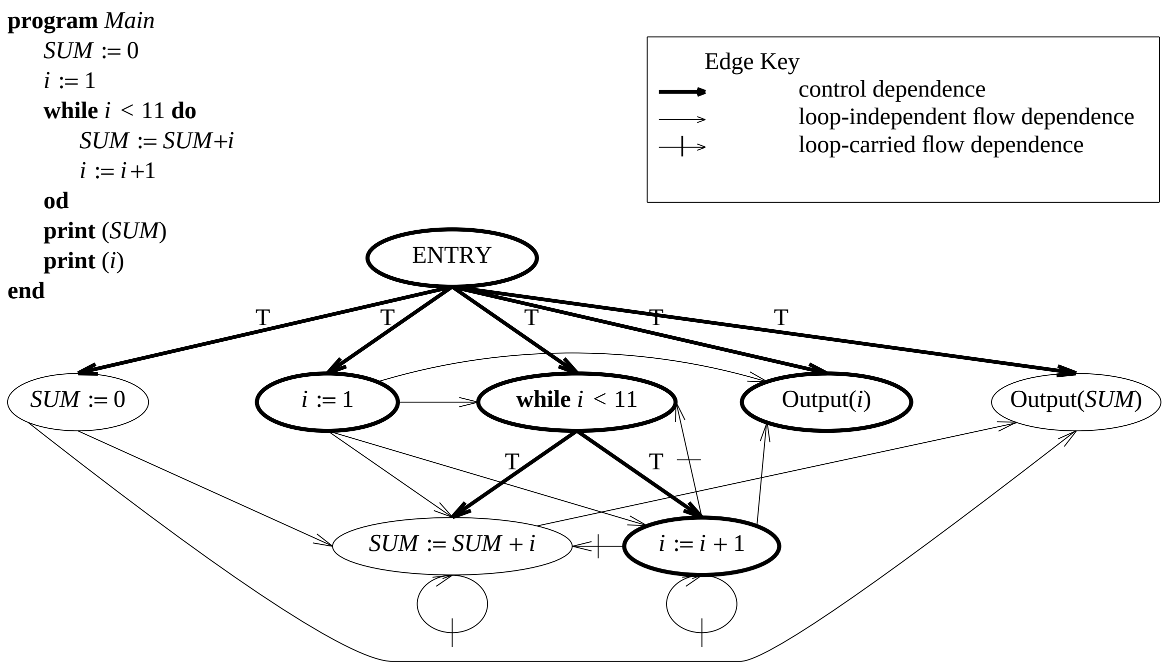

Figure 7: An example program, which sums the integers from 1 to 10 and leaves the result in the variable sum, and its program dependence graph. The vertices of the slice of this PDG taken with respect to the final use of i is shown in bold.

is used in the construction of the PDG with a set of *fall-through* edges, i.e., the lexcial successor of the *goto* in the source text. The fall-through edges captures the requisite control flow when a *goto* statement is deleted from the slice.

Ball and Horwitz [7] describe a similar technique in which jump statements are represented by pseudo-predicate vertices, which always evaluate to **true**. The outgoing control edge labeled **true** is connected to the target of the jump, while a **false** successor is connected to the jump statement's continuation: *i.e*, the statement that would be executed if the jump were a nop.

Harrold, Malloy and Rothermel[38] describe an efficient construction of PDG's that captures exact control dependence (i.e., *gotos*) but uses neither a control flow graph or a post dominator tree as an auxiliary structure. This construction technique improves the methods of [7, 22] for construction of the PDG. During the parse, a *partial control dependence subgraph*, which incorporates exact control is constructed. The partial control dependence subgraph manages control flow by ordering the nodes implicitly during construction or explicitly by the creation of control edges. The presence of gotos does require a minor change to Step 3 of the slicing algorithm (the projection of a program from the sliced PDG). This change ensures that labels of statements not in the slice are include in resulting program if a goto to that label is in the slice.

Another method for computing slices in the presence of arbitrary control flow avoids changing the PDG at the expense of modifying the slicing algorithm use in step (2) [2]. This method maintains two relations: postdominator and lexical-successor. The algorithm computes a slice from the graph use by the HPR algorithm using step (2) and then looks for jump statement not in the slice whose nearest postdominator in the slice is different from the nearest lexical success in the slice. Such statement are then added to the slice. As with the algorithms in [7, 22, 38] and step (3) must be modified to include any necessary labels.

2.2.2 Interprocedural Slicing of Dependence Graphs

Interprocedural slicing as a graph reachability problem requires extending of the PDG and, unlike the addition of data types or unstructured control flow, it also requires modifying the slicing algorithm. The PDG modifications represent call statements, procedure entry, parameters, and parameter passing. The algorithm change is necessary to correctly account for procedure calling context. This section describes the interprocedural slicing algorithm presented in [48], which is based on an extension of the PDG called *the system dependence graph* (SDG).[4] Horwitz et al. [48] introduced the term "system dependence graph" for the dependence graphs that represents multi-procedure programs. The term "system" will be used to emphasize a program with multiple procedures.

The SDG models a language with the following properties:

[4]The term "SDG" is used because the term "PDG"is associated with graphs that represent single procedure programs

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

2. Parameters are passed by value-result.

Techniques for handling parameters passed by reference and for dealing with aliasing as discussed at the end of this section.

Horwitz, et al. [48] make the further assumption that there are no call sites of the form $P(x,x)$ or $P(g)$, where g is a global variable. The former restriction sidesteps potential copy-back conflicts. The latter restriction permits global variables to be treated as additional parameters to each procedure; thus, they are not discussed explicitly.

An SDG is made up of a collection of procedure dependence graphs connected by interprocedural control- and flow-dependence edges. Procedure dependence graphs are similar to program dependence graphs except that they include vertices and edges representing call statements, parameter passing, and transitive flow dependences due to calls (we will abbreviate both procedure dependence graph and program dependence graph by "PDG"). A call statement is represented using a *call* vertex; parameter passing is represented using four kinds of *parameter* vertices: on the calling side, parameter passing is represented by *actual-in* and *actual-out* vertices, which are control dependent on the call vertex; in the called procedure, parameter passing 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 global variable that may be used or modified as a result of the call and for every parameter; formal-out and actual-out vertices are included only for global variables and parameters that may be modified as a result of the call. Interprocedural data-flow analysis is used to determine the parameter vertices included for each procedure [8, 10].

Transitive dependence edges, called *summary* edges, are added from actual-in vertices to actual-out vertices to represent transitive flow dependences due to called procedures. These edges were originally computed using a variation on the technique used to compute the subordinate characteristic graphs of an attribute grammar's nonterminals [48]. Recently, Reps et. al. described a faster algorithm for computing summary edges [84]. A summary edge is added if a path of control, flow and summary edges exists in the called procedure from the corresponding formal-in vertex to the corresponding formal-out vertex. Note that the addition of a summary edge in procedure Q may complete a path from a formal-in vertex to a formal-out vertex in Q's PDG, which in turn may enable the addition of further summary edges in procedures that call Q.

Procedure dependence graphs are connected to form an SDG using three new kinds of edges:

1. a *call* edge is added from each call-site vertex to the corresponding procedure-entry vertex;

2. a *parameter-in* edge is added from each actual-in vertex at a call site to the corresponding formal-in vertex in the called procedure; and

3. a *parameter-out* edge is added from each formal-out vertex in the called procedure to the corresponding actual-out vertex at the call site.

Figure 8 shows an example system and the corresponding SDG. (In Figure 8, as well as in the remaining figures of the paper, the edges representing control dependences are shown unlabeled; all such edges would be labeled **true**.)

Interprocedural slicing can be defined as a reachability problem using the SDG, just as in*tra*-procedural slicing is defined as a reachability problem using the PDG. The slices obtained using this approach are the same as those obtained using Weiser's interprocedural-slicing method [97]. However, his 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 8 from the vertex of procedure $Main$ labeled "$x_{in} := sum$" to the vertex of $Main$ labeled "$i := y_{out}$." However, this path corresponds to procedure Add being called by procedure A, but returning to procedure $Increment$, which is not possible. The value of i after the call to procedure A is independent of the value of sum before the call, and so the vertex labeled "$x_{in} := sum$" should not be included in the slice with respect to the vertex labeled "$i := y_{out}$" Figure 9 shows this slice).

To achieve more precise interprocedural slices, an interprocedural slice with respect to vertex s is computed using two passes over the graph. Summary edges 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. Informally, if s is in procedure P then pass 1 identifies vertices that reach s and are either in P itself or procedures that (transitively) call P. The traversal in pass 1 does not descend into procedures called by P or its callers. Pass 2 identifies vertices in called procedures that induce the summary edges used to move across call sites in pass 1.

The traversal in pass 1 starts from s and goes backwards (from target to source) along flow edges, control edges, call edges, summary edges, and parameter-in edges, but *not* along 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 call, or parameter-in edges. The result of an interprocedural slice consists of the sets of vertices encountered during by pass 1 and pass 2, and the set of edges induced by this vertex set. A worklist algorithm for finding the vertices of an interprocedural slice is stated in [48]. The (full backward) interprocedural slice of graph G with respect to vertex set S, denoted by $Slice(G, S)$, consists of the sets of vertices identified by pass 1 and pass 2, and the set of edges induced by this vertex set.

$Slice(G, S)$ is a subgraph of G. However, unlike intraprocedural slicing, it may be infeasible (*i.e.*, it may not be the SDG of any system). The problem arises when $Slice(G, S)$ includes mismatched parameters: different call-sites on a procedure include difference parameters. There

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

```
procedure A(x, y)
    call Add(x, y)
    call Increment(y)
return
```

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

```
procedure Increment(z)
    call Add(z, 1)
return
```

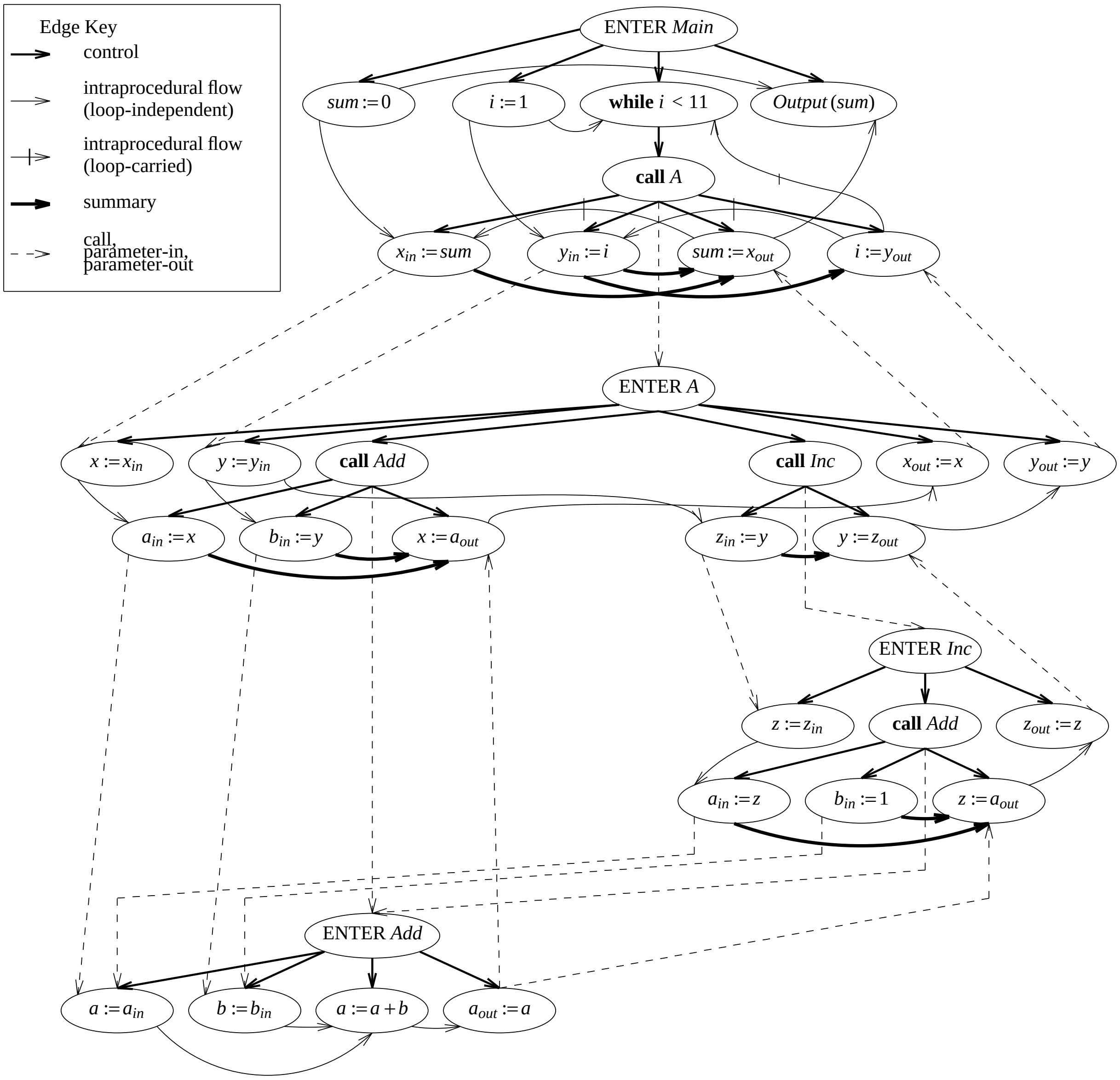

Figure 8: Example system and its SDG.

procedure *Main*
  $i := 1$
  while $i < 11$ **do**
    call $A(i)$
  od
end()

procedure $A(y)$
  call *Increment* (y)
return

procedure *Add* $(a,\ b)$
  $a := a + b$
return

procedure *Increment* (z)
  call *Add* $(z,\ 1)$
return

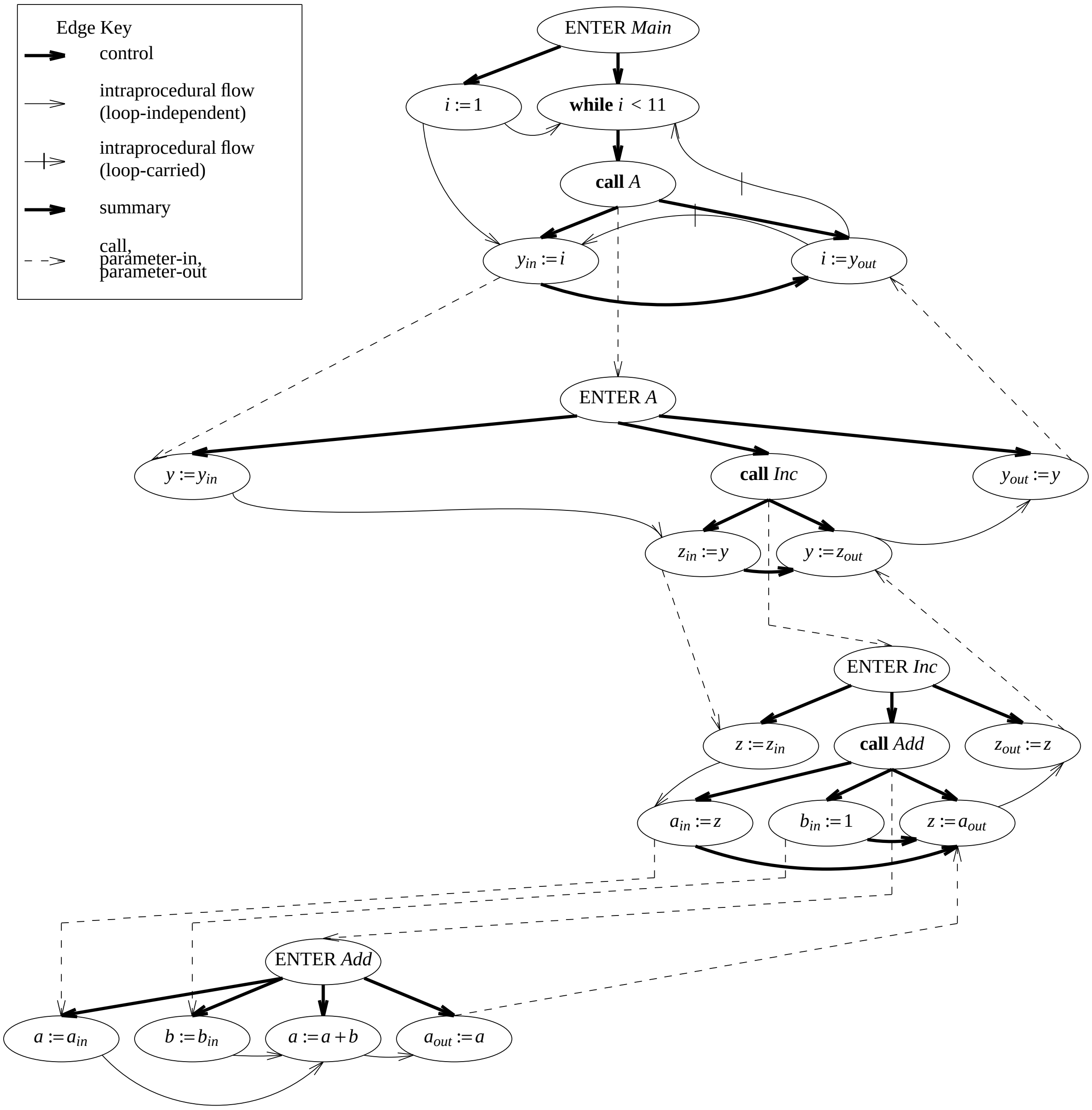

Figure 9: The SDG from figure 8 sliced with respect to the formal-out vertex for parameter z in procedure *Increment*, together with the system to which it corresponds. Note that this slice correctly excludes the vertex labeled $x_{in} := sum$ in figure 8

are two causes of mismatches: missing actual-in vertices and missing actual-out vertices. Making such systems syntactically legal by simply adding missing parameters leaves semantically unsatisfactory systems [19]. In order to include the program components necessary to compute a safe value for the parameter represented at missing actual-in vertex v, the vertices in the Pass 2 slice of G taken with respect to v must be added to the original slice. A Pass 2 slice includes the minimal number of components necessary to produce a semantically correct system. The addition of Pass 2 slices is repeated until no further actual-in vertex mismatches exist.

The second cause of parameter mismatches is missing actual-out vertices. 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 the slice.

A system can now be obtained by projecting the statements of the original system that are in the original slice or added by the above two steps. These statements appear in the same order and at the same nesting level as in the original system. The details of this algorithm are given in [19].

2.2.3 Interprocedural Slicing in the Presence of Call-By-Reference Parameter Passing and Aliasing

The definitions of the system dependence graph and interprocedural slicing assume that parameters are passed by using value-result parameter passing. The same definitions hold for call-by-reference parameter passing in the absence of aliasing; however, in the presence of aliasing, some modifications are required. Two extreme methods for handling aliasing include transforming the system into an aliasing-free system and generalizing the definition of flow dependence. Translation is done by creating a copy of a procedure for each possible aliasing configuration that it may be called under. Because the number of aliasing configurations is potentially exponential, the cost of this transformation may, in the worst case, be exponential in the maximum number of formal parameters associated with any one procedure.

Generalizing the definition of flow dependence makes the pessimistic assumption that any aliases that exist during a particular call to a procedure *may* exist during all calls to the procedure. Such aliases are referred to as *may* aliases. This requires the use of generalized definitions for flow dependence. For example, a flow dependence edge connects vertex v to vertex u iff all of the following hold:

1. v is a vertex that defines variable x.

2. u is a vertex that uses variable y.

3. x and y are potential aliases.

4. Control can reach u after v via a path in the control-flow graph along which there is no intervening definition of x or y.

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

The first solution produces more precise (smaller) slices than the second at the expense of transforming the system. It is possible to consider intermediate solutions to the problem of slicing in the presence of aliasing. Binkley [18] presents an algorithm that is parameterized by a set of aliasing information. The more precise this information, the more precise the slice. In the case of exact information, the same slice is produce as by the transformation approach without replicating procedures. In the case of a maximal approximation (imprecise information), the same slice if obtained as by the generalized dependence approach. In between is a range of possible slices differing only in their precision.

2.3 Dynamic Slicing as a Data-Flow Problem

Korel and Laski[53] introduced the idea of dynamic slicing. Their solution, pattered after Weiser's static slicing algorithm, solves the problem using data-flow equations. A dynamic slice differs from a static slice in that is makes use of information about a particular execution of a program. Thus, a dynamic slice contains "all statements that *actually* affect the value of a variable at a program point for a *particular* execution of the program" rather than "all statements that *may* affected the value of a variable at a program point for any arbitrary execution of the program" [53].

Most dynamic slices are computed with respect to an *execution history* (called a trajectory in [53]). This history records the execution of statements as the program executes. The execution of a statement produces an occurrence of the statement in the execution history. Thus, the execution history is a list of statement occurrences.

Example. Two example execution histories are shown in Figure 10. Superscripts are used to differentiate between the occurrences of a statement. For example, statement 2 executes twice for the second execution producing occurrences 2^1 and 2^2.

Korel and Laski define a dynamic slice, taken with respect to a set of variables V, An Input I, and a point P in the execution history (obtained by running the program on I), as a reduced executable program. The execution history for the execution of this program on input I must satisfy the *slice sub-list* property. Intuitively this property is satisfies if the execution history of the slice is equivalent to the execution history of the original program after removing occurrences of statement not in the slice.

Figure 11 shows the Korel and Laski slice of the program shown in Figure 10 taken with respect to ($\{a\}$, 2, 3^2). Notice that this slice impreciseley includes 3^1 which does not affect the

```
Statement
 Number      Program
   1         read(N)
   2         for i = 1 to N do
   3            a = 2
   4            if c1 then
   5               if c2 then
   6                  a = 4
                   else
   7                  a = 6
                   fi
                fi
   8            z = a
             od
   9         print(z)
```

Execution History 1

Input N=1, *c1* and *c2* both true. $<1^1, 2^1, 3^1, 4^1, 5^1, 6^1, 8^1, 2^2, 9^1>$

Execution History 2

Input N=2, *c1* and *c2* false on the first iteration and true on the second. $<1^1, 2^1, 3^1, 4^1, 8^1, 2^2, 3^2, 4^2, 5^1, 6^1, 8^2, 2^3, 9^1>$

Figure 10: Two execution histories.

read(N)
for $i = 1$ **to** N **do**
    $a = 2$
od

Execution history of slice

$<1^1, 2^1, 3^1, 2^2, 3^2>$

Figure 11: A dynamic slice of the program shown in Figure 10 and its execution history.

computation at 3^2. Why this occurrence is needed in the Korel and Laski framework and how Agrawal and Horgan remove it is discussed at the end of this section.

Two formalize the notion of a dynamic slice, we consider the relationship between the execution histories of the program and its slice (on the same input). The execution history of the slice should be obtained by removing occurrences from execution history of the original program. To do this we must remove occurrences of statement not in the slice and remove all occurrences after the occurrence with respect to which the slice was taken.

Formally, we make use of three the operators:

1. $collect(list, predicate) = [l \in list \mid predicate(l) = \mathbf{true}]$

2. $front(list, l_i) = [l_j \in list \mid j <= i]$

3. $occurrence\_of(l_i, P) = \mathbf{true}$ iff l_i is an occurrence of a statement of P.

Definition 7 *Dynamic Slice. Program P' is the slice of program P taken with respect to (V, I, p) if L and L' are the execution histories for P and P', respectively, p' is the occurrence of p in L', and the following property is satisfied.*

$front(L', p') = collect(front(L, p), \lambda l.occurrence\_of(l, P'))$

EXAMPLE. For the slice taken with respect to ($\{a\}$, 2, 3^1) of the program shown in Figure 10 we have $(< 1^1, 2^1, 3^1, 2^2, 3^2 >, 3^1) = < 1^1, 2^1, 3^1 >$
and

collect (*front* $(< 1^1, 2^1, 3^1, 4^1, 8^1, 2^2, 3^2, 4^2, 5^1, 6^1, 8^2, 2^3, 9^1 >, 3^1)$, $\lambda l.occurrence\_of(l, P'))$ = *collect*$(< 1^1, 2^1, 3^1 >$, $\lambda l.occurrence\_of(l, P')) =< 1^1, 2^1, 3^1 >$
While for the slice with respect to ($\{a\}$, 2, 3^2)

front $(< 1^1, 2^1, 3^1, 2^2, 3^2 >, 3^2) =< 1^1, 2^1, 3^1, 2^2, 3^2 >$.

and

collect$(front(< 1^1, 2^1, 3^1, 4^1, 8^1, 2^2, 3^2, 4^2, 5^1, 6^1, 8^2, 2^3, 9^1 >, 3^2), \lambda l.occurrence\_of(l, P'))$ = *collect*$(< 1^1, 2^1, 3^1, 4^1, 8^1, 2^2, 3^2 >$, $\lambda l.occurrence\_of(l, P')) =< 1^1, 2^1, 3^1, 2^2, 3^2 >$.

Dynamic slices are computed from three data flow relations computed from the program. The first two capture the flow and control dependences. Unlike static slicing, these definitions refer to the execution history; thus, they capture dependences that actually happened in the particular execution of the program rather than dependences that may occur in some execution of the program as is the case with static slicing. The final relation is explained below.

The first definition captures flow dependences that arise when one occurrence represent the assignment of a variable and another a use of that variable. For example, in Figure 10 when *c1* and *c2* are both false, there is a flow dependence from s_3 to s_8.

Definition 8 *Definition-Use (DU). v^i DU u^j $\Leftrightarrow$ v^i appears before u^j in the execution history and there is a variable x defined at v^i, used at u^j, but not defined by any occurrence between v^i and u^j.*

The second definition captures control dependence. The only difference between this definition and the static control dependence definition is that multiple occurrence of predicates exist. Consequently, care must be taken to avoid a statement earing dependent on a predicate from an earlier iteration of a loop.

Definition 9 *Test Control (TC).* $v^i\, TC\, u^j \Leftrightarrow$ *u is control dependent on v (in the static sense) and for all occurrences w^k between v^i and u^j, w is control dependent on v.*

The definition of a dynamic slice requires the use of $front(L', p') = collect(front(L, p), \lambda l.occurrence\_of(l, P'))$. The above two relations are insufficient to ensure this. For example, the slice obtained from the program in Figure 10 taken with respect to ($\{a\}$, 2, 3^2) using only the above two relations is $< 1^1, 2^1, 2^2, 3^2 >$, which omits 3^1. In this example the omission would be benign, but in other examples the slice would not have the correct execution history. Korel and Laski solve this problem by including the following relation. Note that this produces a conservative solution to the problem. That is, some dynamic slices are bigger than they need to be.

Definition 10 *Identity Relation (IR).* $v^i\, IR\, u^j \Leftrightarrow v = u$.

The relations DU, TC, And IR are used to formulate the algorithm for a dynamic slicing by computing an ever growing approximation to occurrences in the slice. Let E be the execution history produces for program P when run on input I. In the following two equations only the $front$ of E is of interest to the dynamic slice, since only these occurrences may affect the point where the slice originates.

The first approximation to the slice taken with respect to (V, I, l^i) contains the direct control and data influences:

$S^0 = \{v^j \in front(E, l^i) \mid v^j\, TC\, l^i\} \cup \{v^j \in front(E, l^i) \mid x \in V \wedge x$ is defined at v^j but not defined by any occurrence between v^j and l^i.

The slice is found by iterating the following until a fixed point is reached. $S^{i+1} = S^i \cup \{v^j \in front(E, l^i) \mid \exists u^k \in S^i$ s.t. $v^j\, (DU \cup TC \cup IR)\, u^k\}$. The above iteration must terminate since E is finite and bounds each S^i. To complete the slice it is necessary to include the occurrence with respect to which the slice was taken. Finally, the sliced program P' is obtained by projecting out of P those statement that have occurrences the fixed point solution.

Example. The dynamic slice of Execution History 2 from Figure 10 taken with respect to ($\{a\}$, 2, 3^1) is computed as follows

$DU = \{(1^1, 2^1)(3^1, 8^1)(6^1, 8^2)(8^2, 9^1)\}$

$TC = \{(2^1, 3^1)(2^1, 4^1)(2^1, 8^1)(2^1, 3^2)(2^1, 4^2)(2^1, 8^2)(2^2, 3^2)(2^2, 4^2)(2^2, 8^2)(4^2, 5^1)(5^1, 6^1)\}$

$IR = \{(2^1, 2^2)(2^1, 2^3)(2^2, 2^3)(3^1, 3^2)(4^1, 4^2)(8^1, 8^2)\}$

$S^0 = \{2^1\}$

$S^1 = \{1^1, 2^1\}$

$S^2 = \{1^1, 2^1\} = S_1$; thus the iteration ends.

$S = \{3^1\} \cup \{1^1, 2^1\} = \{1^1, 2^1, 3^1\}$.

Similar to static slicing, handling unstructured control flow requires modifying the definition of *control dependent* on used in the definition of TC. Unlike static slicing, handling different data types can be done more precisely in a dynamic slice. First consider arrays. Because the subscript value is known at run time, the ambiguity of array accesses in static slicing is gone. The only complication this raises is that different occurrences of a statement may assign different variables (for simplicity, we consider each array element a different variable). Records are similarly handled by treating each component as a separate variable. Finally, consider dynamically allocated memory. This can also be handled by giving each newly allocated block of memory a pseudo name. At the end of the next section we describe how Agrawal and Horgan replace the notion of (pseudo) variable name to improve dynamic slicing in debugging.

2.4 Dynamic Slicing as a Graph-Reachability Problem

Agrawal and Horgan presented the first algorithms for finding dynamic slices using dependence graphs [5]. We first consider two simple dynamic slicing algorithms that are imprecise. We then consider two exact algorithms that differ only in their space complexity. The initial description of all four algorithms assumes scalar variables. The extensions necessary to handle complex data structures including, for example, (unconstrained) pointers, arrays, records, and unions are considered at the end of this section

The first two algorithms both mark some portion of a PDG as "executed." The first marks executed vertices and the second marks executed edges. Both algorithms suffer because they summarize in one place (a vertex or an edge) all the times a statement or dependence edge between statements was executed. The first algorithm initially labels all vertices in the PDG as "unexecuted." As the program runs, whenever a statement is executed its corresponding vertex is marked as "executed." After execution the slice is computed by applying the static slicing algorithm restricted to only marked vertices (and the edges that connect them). Thus, unexecuted vertices will not be in the dynamic slice.

As illustrated in Figure 12 by Slice 1, this approach produces imprecise slices because it is possible for a vertex to be marked executed even though it does not actually affected a given computation. For Slice 1, s_3 is executed, but the assignment at s_6 overwrites the value written to a. Therefore, s_3 need not be in the dynamic slice.

The second algorithm marks "executed edges." This produces more precise answers by leaving out statements that were executed, but had no influence on the variables in the slice (the corresponding edges were not "executed." It's imprecision comes in loops where different iterations execute different edges. In Slice 1 shown in Figure 12, marking edges allows the algorithm to leave out s_3, because the flow dependence edge from s_3 to s_8 is never executed. However, for the 2, this edges is executed during the first iteration of the loop. Since the second iteration overwrites the value of a, s_3 need not be in the dynamic slice. The problem is that different iterations require marking different edge *occurrences* . Summarizing all occurrences on

Slice 1: slice on z at s_9 for the input N=1, where *c1* and *c2* both true:

static slice contains s_1–s_9

dynamic slice (marking nodes) contains s_1–s_6, s_8, s_9

dynamic slice (marking edges) contains s_1, s_2, s_4–s_6, s_8, s_9

Slice 2: slice on z at s_9, for input N=2, where *c1* and *c2* false on the first iteration and true on the second:

static slice contains s_1–s_9

dynamic slice (marking nodes) contains s_1–s_6, s_8, s_9

dynamic slice (marking edges) contains s_1–s_6, s_8, s_9

Figure 12: Two dynamic slices of the program shown in Figure 10 that show the limitation of the first two dynamic slicing algorithms.

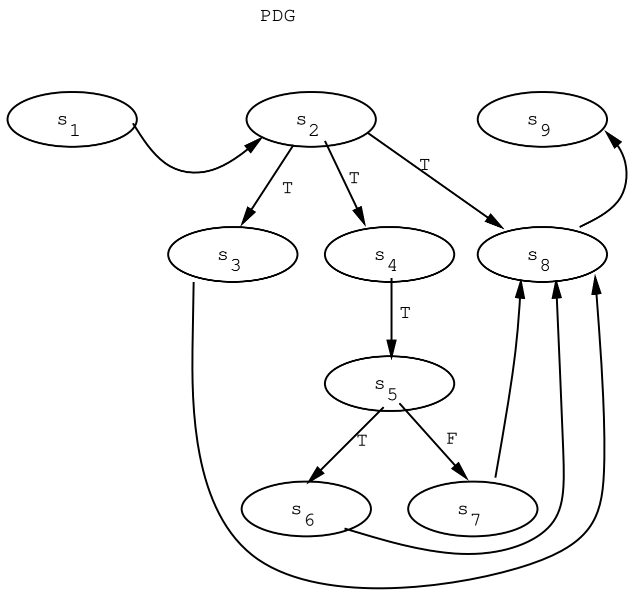

Figure 13: The PDG for the program shown in Figure 10

a single edge introduces imprecision.

Example. Figure 12 compares static slicing with these two dynamic slicing algorithms for two different inputs. (Figure 13 shown the PDG for this program). The static slice taken at (the vertex representing) statement s_9 with respect to z contains the entire program because all three assignments to a may reach the use at statement s_8. The node-marking dynamic slicing algorithm produces a smaller slice for both inputs because s_7 does not execute in either case. These slices, however, unwantedly include s_3 which does execute, but which does not affect the computation of z. The edge-marking algorithm correctly handles this in Slice 1, because the flow dependence edge from s_3 to s_8 is not executed. However, the weakness of the edge marking algorithm is shown by Slice 2: Although the assignment at s_2 does no affect the output value of z, this flow edges is executed by the first iteration of the loop. Because it is marked executed, s_2 is included in the slice.

In order to omit statements in loops like s_3, it is necessary to have multiple occurrences of the vertices and edges in the loop. This is captured by the *execution history* of a program for a given input. For example, Slice 2 of the program in Figure 12 produces the execution history $< 1,\ 2^1,\ 3^1,\ 4^1,\ 8^1,\ 2^2,\ 3^1,\ 4^2,\ 5^2,\ 6^2,\ 8^2,\ 2^3,\ 9 >$ where the superscripts are used to denote different executions of the same statement.

A dynamic slice can now be defined in terms of a variable, an execution history, and an

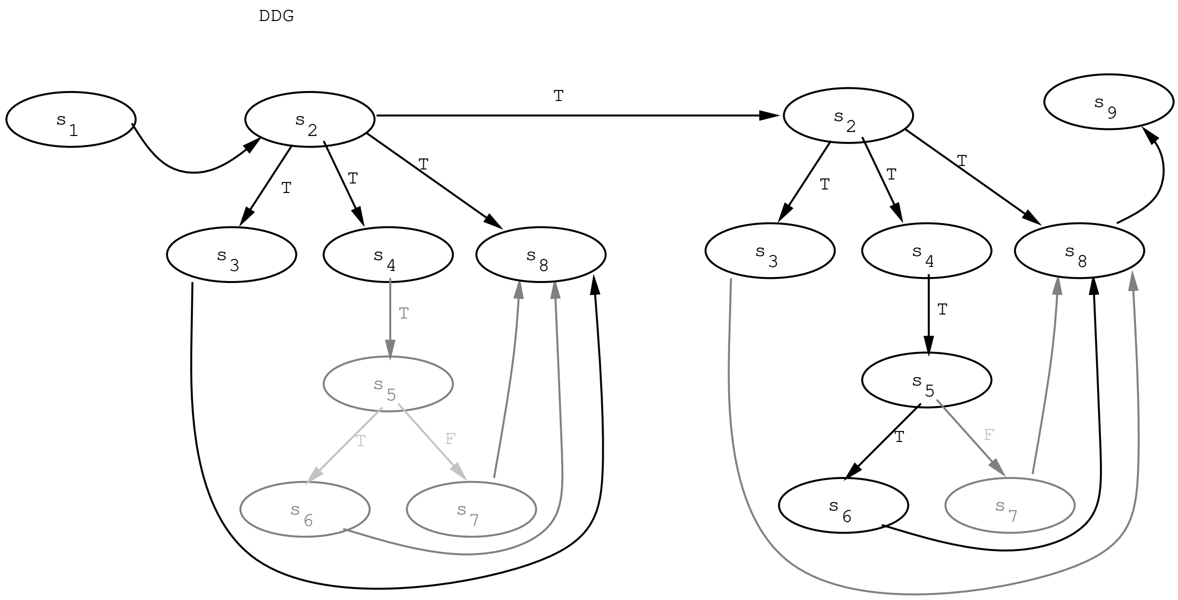

Figure 14: The DDG for the Slice 2 in the examle shown in Figure 12

occurrence of a statement in the execution history. The slice contains only those statements whose execution had some effect on the value of the variable at the occurrence of the statement in the execution history. To obtain this slice a *Dynamic Dependence graph* (DDG) is produced from the execution history. A dynamic dependence graph has a vertex for each occurrence of a statement in the execution history and contains only executed edges. Because a DDG separates the occurrences of the executed vertices and edges, a vertex in a loop to have multiple occurrences, which can have different incoming dependence edges. This removes the imprecision in the two simple approaches.

Control dependence edges of the DDG are copies of the control dependence edges from the static dependence graph. There is a single dynamic flow dependence for each variable v referenced at a node n. The source of this edge is computed from the Dynamic Reaching Definitions (DRDs), which are in turn computed from the execution history as follows:

$DRD(var, empty) = \emptyset$.

$DRD(var, < previous\_history \mid last\_node >) =$

$$\begin{cases} \{last\_node\} & if\ var \in def(last\_node) \\ DRD(var, previous_history) & otherwise \end{cases}$$

EXAMPLE. Figure 14 shows the DDG for the Slice 2 of the program shown in Figure 12. This graph contains two occurrences of s_8. The first has an incoming flow dependence edge from s_3 while the second has an incoming flow dependence edge from s_6. Since the only occurrence of s_9 has an incoming edges from the second occurrence of s_8, s_6 and not s_3 are included in the slice.

The disadvantage of this approach is its space requirement. Since a loop may execute an unbounded number of times, there may be an unbounded number of vertex occurrences. This is clearly unacceptable and, it turns out, unnecessary. A dynamic slice is a subset of the original program; consequently, there can be only a finite number of slices (subsets). The final dynamic slicing algorithm, which computes the same slices as the third algorithm, exploits this fact to limit the size of the dynamic dependence graph.

The fourth algorithm, considers each new vertex occurrence v as the DDG is built. If a suitable vertex occurrence u exists in the DDG then v is merged into u. Vertex occurrence u is suitable if its slice is the same as the slice of the new vertex occurrence. That is the statements whose occurrences affect u are the same as the statements that affect v. Note that since the slice is a projection of the DDG back on to the original program, it is possible to combine vertices that are not created from the same statement.

The algorithm maintains three sets: $DefinitionNodes$, which maps a variable's name to the last vertex occurrence that defines it; $PredicateNode$, which maps a control statement to its most recent occurrence in the DDG; and $ReachingStatements$, which, for each node n, contains all the statements of the original program with an occurrence that reaches n. The slice with respect to a occurrence of statement s is simply those statements in $ReachingStatements(s)$, when the occurrence of s is created.

2.4.1 Dynamic Slicing of Programs with Unconstrained Pointers

Agrawal and DeMillo extended the above notion of dynamic slicing to programs that contain unconstrained (C like) pointers. The technique applies also to arrays, constrained (Pascal like) pointers, records, and other data structures [4].

What makes unconstrained pointers difficult is the loss of the 1-1 mapping from variables assigned and used and from *memory locations* assigned and used. This is no longer true in the presence of more complexed data structures. While aggregates such as records can be handled by treating each field as a separate variable, and even array elements can be treated as separate elements, since dynamic slicing has access to the run-time index value, pointers presents a particularly difficulty problem. If we ignore "pointer laundering" through variant records, *constrained* pointers, such as those found in Pascal, always point to a nameable object of a particular type. Special names are given to objects allocated from the heap. As with records and arrays, run-time information allows us to identify the particular object pointed to by a given pointer. In contract, *unconstrained pointers* , such as those found in C, may use or modify arbitrary (unnamed) memory locations.

This forces us to generalize first the notion of a variable (*var* in the proceeding definition of a Dynamic Reaching Definition) to *memory cells.* Each cell is composed of a starting address and a length. A flow dependence exists between a node that modifies $cell_1$ and a node the reference $cell_2$ if $cell_1$ and $cell_2$ overlap. This occurs when there is a non-empty intersection of the set of addresses $cell_1.start, ..., cell_1.start + cell_1.length$ and $cell_2.start, ..., cell_2.start + cell_2.length$. For example, if $cell_1$ represents an assignment to a structure and $cell_2$ represents a use of a field in this structure then the two cells overlap. Notice that an assignment to a field of the structure may only modify part of the cell (the same is true of assignments to an element of an array). In the following definition, the function $PreCell(cell_1, cell_2)$ denotes the part of $cell_1$ before $cell_2$. Likewise, $PostCell(cell_1, cell_2)$ denotes the part of $cell_1$ after $cell_2$.

Definition 11 *DRD in the presence of unconstrained pointers. Let* $cell' = def(last\_node)$.

$DRD(cell, empty) = \emptyset$

$DRD((address, 0), history) = \emptyset$

$DRD(cell, < previous\_history \mid last\_node >) =$

$$\begin{cases} DRD(cell, previous\_history) & if\ cell \cap cell' = \emptyset \\ last\_node \cup DRD(PreCell(cell, cell'), previous\_history) & \\ \quad \cup DRD(PostCell(cell, cell'), previous\_history) & otherwise \end{cases}$$

This technique works well when dealing the kind of error shown in the following code (note that different compilers may lay memory out differently):

```
{
    int i, j, a[4];

    j = 2;
    for (i=0; i<=4; i++)
       a[i] = 0;         /* incorrectly assigns j=0 when i = 4 */

    print j;
}
```

When $i = 4$, the cell defined by `a[i] = 0` and the cell for j overlap (the are identical in this example). A static slice with respect to j at the end of the program would only include the assignment $j = 2$. However, the dynamic slice using the above definition of dynamic data definition includes the assignment "$a[i] = 0$" and the enclosing for loop. An indication that j is getting clobbered by the loop.

2.5 Alternative Approaches to Computing Program Slices

2.5.1 Denotational Program Slicing: Functional Semantics

Hausler [39] has developed a denotational approach for slicing structured programs. This approach uses the functional semantics of the "usual" restricted structured language; one with scalar variables, assignment statements, if-then-else statements, and while statements. The Mills' semantics [60] is defined for the language; the meaning of the program is determined by a mapping from the variables to values.

In the Mills' semantics, programs are constructed by composition of statements (*i.e.* functions from state to state). This permits the definition of a *sliceable subprogram*: any sequence of statements with the last j statements deleted. Thus slices can be taken at arbitrary points in the program.

The function

$$\alpha : statements \times variables \rightarrow variables$$

captures the relevant sets. While the function

$$\delta : statements \times variables \rightarrow statements$$

captures the slice. These two functions are defined semantically for each language construct. This yields a precise "Millsian" slice; moreover, gives primitive recursive finite upper bounds on the number of times repetitive statements must be iterated to obtain the slice. These bounds are linear in the number of assignment statements in the program.

2.5.2 Denotational Program Slicing: Descriptive Semantics

Venkatesh [91] also looked at denotational slicing with the aim of separating semantically based definitions of a program slice from the justification of the algortihm to compute the slice. He has constructed a three dimensional framework for the comparison and analysis of the definitions of a slice:

1. Executable or Closure[5]
2. Forward or Backward
3. Static or Dynamic

and then gives semantic definitions of the 8 combinations.

In [91] the idea of "contamination" was introduced as an extension to the usual semantics of the language in order to clarify the differences of the concepts. This notion is further elaborated on in [34] as a form of error propagation to slice functional programs. In both instances,

[5] "Closure" comes from the graph-theoretic method used to compute the slice.

contaminated values are tracked via the semantics; in the end all contaminated expressions are those that are in the forward slice.

Tip [89] gives an alternative characterization using 4 criteria:

1. Computation Method: Data-Flow, Functional, or Graph-Reachability
2. Interprocedural Solution or not
3. Control Flow: Structured or Arbitrary
4. Data types: Scalars, Composites, Pointers.

2.5.3 Information Flow Relations

Bergeretti and Carré [13] construct three binary information flow relations on a structured subset of a Pascal-like language. These relations are associated with each program statement (and hence inductively over the entire language). The first relation, λ, over $V \times V$, where V is the set of variables of the program, associates values of program variables on entry to a statement, S, with values of variables on exit from S. This can be loosely interpreted as "the value of v on entry to S *may be used* in the evaluation of the expression e in S." This relation is a formalization of the results of Denning and Denning [25] in secure information flow.

The second relation, ρ, also over $V \times V$, associates values of variables on entry to statement S with the value(s) of the expression parts in S. That is, "the value of v on entry to S *may be used* in obtaining the value w on exit from S." The entry value of v may be used in obtaining the value of some expression e in S, that in turn may be used in obtaining the exit value of w.

The third relation, μ, over $E \times V$ where E is the set of expressions in the program, associates an expression e with a variable v for statement S iff "a value of expression e in S *may be used* in obtaining the value of the variable v on exit from S." For example, if S is the assignment statement "$a := b + c$" then

$$
\begin{aligned}
\lambda(S) &= \{(b, a), (c, a)\} \\
\rho(S) &= \{(b, a), (c, a)\} \cup \{(v, v) - (a, a) \mid v \in V\} \\
\mu(S) &= \{(b + c, a)\}
\end{aligned}
$$

Transitive closure of the μ relation permits construction of a *partial program* associated with a variable. This partial program is a program slice taken with respect to the variable at the last statement of the program. Relation μ is related to the edges of a dependence graph. For example, it need only be computed once after which a program slice can be computed in linear time.

2.5.4 Parametric Program Slicing

Parametric program slicing [27] uses graph rewriting to compute *constrained* slices. The constraint refers to the amount of input available. A fully constrained slice (where input has a fixed constant value) is a dynamic slice, while a fully unconstrained slice is a static slice. A partially constrained slice is computed when some inputs have known values. The resulting slice is smaller than a static slice, but does not require complete execution of the program as with a dynamic slice.

Constrained slices are computed using a term graph rewriting system [9]. As the graph is rewritten, modified terms and sub-terms are tracked. As a result, terms in the final (completely rewritten) graph can be tracked back to terms in the original graph. This identifies the slice of the original graph that produced the particular term in the final graph. A minimal amount of syntactic post processing is necessary to convert this "term slice" into a syntactically valid sliced program.

2.5.5 Dynamic Slicing using Dependence Relations

Gopal describes a technique for computing dynamic slices using *dependence relations* [35]. This approach abstracts from the program three relations at each statement S:

1. S_v: the dependence of statement S on the input value of variable v.
2. v_S: the dependence of output variable v on statement S.
3. v_u: the dependence of output variable v on the input value of variable u.

Rules are given for different statement kinds (*e.g.*, assignment and conditional statements) and for sequences of statements. For example, the rules for the composition of statements S^1 and S^2 are as follows ($\circ$ represents the composition operator):

1. $S_v \equiv S_v^1 \cup (S_v^2 \circ v_u^1)$
2. $v_S \equiv v_S^2 \cup (v_u^2 \circ v_s^1)$
3. $v_u \equiv (v_s \circ S_v) \cup (S_v^2 \cap S_v^1)$

2.5.6 Parallel Slicing

Danicic, et al. [37] introduce a parallel algorithm for computing backward, static slices. This is accomplished by converting the CFG into a network of concurrent processes. Each process sends and receives messages that names the relevant sets of variables. The significant contribution of this work is that the algorithm outputs the *entire set* of criteria which would yield the computed slice. Thus the set of all criteria of a given program is partitioned into equivalence classes, with the slice itself used as the partitioning relation.

2.5.7 Value Dependence Graphs

The *value dependence graph*, (VDG) [92], is a data flow-like representation that evolved from an attempt to eliminate the control flow graph as the basis of the analysis phase and using *demand* dependences instead. The VDG is a representation that is independent of the names of values, the locations of the values, and when the values' are computed. It is a functional representation that expresses computation as value flow. A value is computed *if it is needed.* Thus, VDG requires explicit representation of stack and heap allocators, I/O ports, etc, so that value can be obtained. Loops are represented as function calls, so no backward pointing edges are required, as in the CFG. The VDG has two advantages that make it suitable for program slicing: all operands are *directly* connected to the inputs (via the functional semantics) and the computation is expressed as value flow. So a single VDG represents the slices for every possible computation. The drawback to this approach is that points of computation are lost; values are the only sliceable objects.

3 Applications of Program Slicing

This section describes how program slicing is used in a collection of application domains. In applying slicing to these domains several variations on the notions of program slicing as describe in Section 2 are developed. The order of this section in not necessarily the historical order in which the problems were addressed. Rather, they are organized to facilitate presentation. For example, the differencing techniques described in Section 3.1 grew out on the integration work described in Section 3.2.

3.1 Differencing

Programmers are often faced with the problem of finding the differences between two programs. Algorithms for finding *textual* differences between programs (or arbitrary files) are often insufficient. Program slicing can be used to identify *semantic* differences between two programs. There are two related programs differencing problems:

1. Find all the components of two programs than have different behavior.

2. Produce a program that captures the semantic differences between two programs.

Dependence graphs solutions to both problems have been given. The only reason for not using the data-flow approach is efficiently; the solutions require multiple slices of the same program, which can be done in linear time using dependence graphs.

For programs *old* and *new*, a straightforward solution to Problem 1 is obtained by comparing the backward slices of the vertices in *old* and *new*'s dependence graphs G_{old} and G_{new}. Components whose vertices in G_{new} and G_{old} have isomorphic slices (see [45] for a definition of isomorphic slices) have the same behavior in *old* and *new* [86]; thus, the set of vertices from G_{new} for which there is no vertex in G_{old} with an isomorphic slice *safely approximates* the set of components *new* with changed behavior. This set is a *safe* as it is guaranteed to contain all the components with different behavior. It is (necessarily) an *approximation* because the exact differencing problem is unsolvable.

We call the vertices in G_{new} with different behavior than in G_{old} the set of *affected points*. The complexity of the straight forward solution for finding affected points outlined above is cubic in the size of G_{new} (slice isomorphism can be determined in linear time [45]). This set can be efficiently computed in linear time using a single forward slice starting from the set of *directly affected points*: those vertices of G_{new} with different incoming dependence edges than in G_{old} [21].

A solution to the second differencing problem is obtained by taking the backward slice with respect to the set of affected points. For programs with procedure and procedure calls, two modifications are necessary: First, the techniques described at the end of Section 2.2.2 are required to ensure the resulting program is executable. Second, this solution is overly pessimistic: consider a component c in procedure P that is called from two call-sites c_1 and c_2. If c is identified as an affected point by a forward slice that enters P through c_1 then, assuming there is not other connection, we want to include c_1 but not c_2 in the program that captures the differences. However, the backward slice with respect to c would include both c_1 and c_2. A more precise solution can be obtained by using a combination of the individual interprocedural slicing passes describes in Section 2.2.2 [21].

3.2 Program Integration

The program integration operation concerns the problem of merging program variants [15, 21, 43]. Given a program $Base$ and two variants, A and B, each created by modifying separate copies of $Base$, the goal of program integration is to determine whether the modifications interfere, and if they do not, to create an integrated program that incorporates both sets of changes as well as the portions of $Base$ preserved in both variants. The need for program integration arises when a system is "customized" by a user and simultaneously upgraded by a maintainer, and the user desires a customized, upgraded version; or when several versions of a program exist and the same enhancement or bug-fix is to be made to all of them.

The integration algorithm uses program differencing to identify the changes in Variants A and B with respect to $Base$. Preserved components are those components that are not affected in A or B. This set is safely approximated as the set of components with isomorphic slices in $Base$, A, and B. A merged program is obtained by taking the graph union of the (dependence graph for) the differences between A and $Base$, the differences between B and $Base$, and the preserved components. This program is then checked for interference. Interference exists if the changes in Variant A and Variant B are incompatible. If there is no interference a merged program is produced that captures the changed *behavior* of A and B along with the preserved *behavior* of all three programs.

While it is NP-hard[43], an important property of the algorithm is that it is semantics-based. An integration tool makes use of knowledge of the programming language to determine whether the changes made to $Base$ to create the two variants have undesirable semantic interactions; only if there is no such interference will the tool produce an integrated program. The algorithm also provides guarantees about how the execution behavior of the integrated program relates to the execution behaviors of the base program and the two variants.

3.3 Software Maintenance

Software maintainers are faced with the upkeep of programs after their initial release and face the same problems as program integrators: understanding existing software and making changes without having a negative impact on the unchanged part. A new kind of slice, called a decomposition slice [32], is useful in making a change to a piece of software without unwanted side effects.

While a slice captures the value of a variable at a particular program point (statement), a decomposition slice captures all computations of a variable and is independent of program location. A decomposition slice is useful to a maintainer when, for example, variable v is determined to have a value to be changed. A differencing tool based on decomposition slicing, called the Surgeon's Assistant [28, 32], partitions a program into three parts (assume the computation of variable v is to be changed):

Independent Part. Statements in the decomposition slice taken with respect to v that **are not** in any other decomposition slice.

Dependent Part. Statements in the decomposition slice taken with respect to v that **are** in another decomposition slice.

Compliment. Statements that are not independent (*i.e.*, statements in some other decomposition slice, but not v's).

Variables can be similarly categorized:

Changeable. *All* assignments to the variable are within the independent part.

Unchangeable. At least one assignment to the variable is in a dependent part. If the maintainer modifies this statement, the new value will flow out of the decomposition.

Used. Variables used in the compliment but not in the independent or dependent parts. The maintainer may *not* declare new variables with these names.

For a maintainer trying to change the code, only independent and dependent statements (*i.e.*, the decomposition slice taken with respect to v) are of interest. Furthermore, the Surgeon's Assistant only allows modifications of the independent part and changeable variables (and newly declared variables). The advantage of this approach is that after making a modification only the independent and deppendent parts need be tested. The complement is guaranteed to be unaffected by the change; testing on the complement is unnecessary[31]. Finally, a program is formed by merging the modified independent part and the unchanged compliment. This can be done in linear time[30], without adverse side affects to computations in the compliment. The result is a modified and tested program.

3.4 Testing

Software maintainers are also faced with the task of regression testing: retesting software after a modification. This process may involve running the modified program on a large number of test cases, even after the smallest of changes. Although the effort required to make a small change may be minimal, the effort required to retest a program after such a change may be substantial. Several algorithms based on program slicing have been proposed to reduce the cost of regression testing. While decomposition slicing eliminates the need for regression testing on the complement, there still may be a substantial number of tests to be run on the dependent, independent and changed parts. Slicing can be used to reduce the number of these tests.

The following algorithms assume programs are tested using test data adequacy criterion: a minimum standard that a test suite (a set of test cases) must satisfy. An example is the *all-statements* criterion, which requires that all statements in a program be executed by at least one test case in the test suite. Satisfying an adequacy criterion provides some confidence that the test suite does a reasonable job of testing the program [11, 79, 20].

Gupta *et. al.*, present an algorithm for reducing the cost of regression testing that uses slicing to determine components affected transitively by an edit at point p [36]. They consider a variety of different types of edits (statement addition, statement deletion, modification of the variables used in a statement, etc.) While some of these require simplified versions of the following procedure, in general two slices are used. The first slice is a backward slice from p. Definitions in this slice of variables used at p are recorded. The second slice is a forward slice also starting from p. Uses in this slice of variables defined at p are recorded. Def-Use pairs from a definition in the first slice to a use in the second are potentially affected by the change and must be retested.

Bates and Horwitz present test case selection algorithms for the all vertices and all flow-edges test data adequacy criterion [11]. The key to their algorithm is the notion of *equivalent execution patterns*. Two program components with equivalent execution patterns are executed by the same test cases [11]. Consider, for example, a component *old* from a tested program and component *new* of a modified version of this program. If test case t tests *old*, and *old* and *new* have equivalent execution patterns then test case t is guaranteed to test *new*. No new test case need be devised (even if *new* does not exist in the original program). The algorithms only selects tests that test changed portions of the modified program.

Components with equivalent execution patterns are identified using a new kind of slice called a *control slice*. A control slice, which is essentially a slice taken with respect to the control predecessors of a vertex, includes the statements necessary to capture "when" a statement is executed without capturing the computation carried out at the statement.

The notion of equivalent execution patterns is to strong in the presence of procedures and procedure calls because it does not separate different calling-contexts (*i.e.*, different chains of call sites). Consider a simple program with two procedures $main$ and P where $main$ calls P. If another call to P is added to $main$ then the control slice for any component in P will include this new call and therefore cannot be isomorphic with any control slice from the original program. Consequently two such components cannot have equivalent execution patterns.

Calling context is more accurately accounted for by replacing equivalent execution patterns with the weaker notions of *common execution patterns* [20]. Components with common execution patters have equivalent execution patterns in some calling context. These components are identified using another kind of slice called a *calling-context slice*, which applies the second pass of backward slicing algorithm described in Section 2.2.2 "back" through the sequence of call-sites that make up a calling-context.

Program differencing can be used to further reduce the cost of regression by reducing the size of the program that the tests must be run on [17]. For a small change, the program produced using the program differencing techniques described in Section 3.1 is considerably smaller and consequently requires fewer resources to retest, especially when run on the reduced test set produced by any of the above algorithms.

3.5 Debugging

Program slicing was discovered as an operation performed by experienced programmers when debugging code [95, 93]. Programmers, given a piece of code with a bug in it, were asked to describe the code after removing the bug. They could reproduce certain "parts" of the code almost exactly, while others they could not. These parts were not continuous blocks of text (*e.g.*, files, procedures, or functions), but rather they were what we now call program slices. Formalization of this debugging activity lead to the first algorithms for program slicing [93, 94].

Turning this around, a tool that computes program slices is a valuable aid in debugging. It allows the programmer to focus attention on those statements that contribute to a fault. In addition, hilighting a slice assists in uncovering faults caused by a statement that should be in a slice but is not.

Several kinds of slices are useful in debugging. First, dynamic slicing is one variation of program slicing introduced to assist in debugging [53, 54]. When debugging, a programmer normally has a test case on which the program fails. A dynamic slice, which normally contains less of the program than a static slice, is better suited to assist the programmer in locating a bug exhibited on a particular execution of the program. As seen in section 2.4.1, dynamic slicing can even assist in finding bugs caused by invalid pointers or array subscripts.

Slicing is also useful in algorithmic debugging, which applies the following algorithm to the debugging process: starting from a external point of failure (*e.g.*, an arrant output value), the debugging algorithm localizes the bug to within a procedure by asking the programmer a series of questions. These questions relate to the expected behavior of a procedure. For example, "should $add(4, 2)$ return 6?" Based on these answers, procedures that have the expected output are treated as working, while procedures that produce unexpected answers are "opened." This means the debugger attempts to determine if the calls in the procedure produce expected results. Algorithmic debugging terminates ends at a procedure with no calls or in a procedure in which all the calls produce the expected output.

One draw back of algorithmic debugging is its asking questions about procedures that do not effect a buggy result. Program slicing can be of assistance here [50]: any call not in the slice with respect to the buggy output can be ignored; it cannot affect the buggy result. Further, parameters that are not in the slice, even for a call that is, can also be ignored.

Debugging was also the motivation for *program dicing* and latter *program chopping*. Dicing uses the information that some variables fail some tests, while other variables pass all tests, to automatically identify a set of statements likely to contain the bug [66]. A program dice is obtained using set operations on one or more backward program slices. Some dices are more interesting than others. The interesting ones include the intersection of two slices, and the intersection of slice A with complement of slice B. The first dice, which identifies common code, is helpful in locating bugs when two computations are both incorrect assuming all incorrectly computed variables are identified and no program bug masks another bug [66]. This dice is also useful in ensuring software diversity in safety critical systems. If for example, the computation of *trip-over-temperature-sensor* and *trip-over-pressure-sensor* include the same computation (often a function call) then this computation is in the dice taken with respect to the two *trip* signals. Such a computation is of interested in safety critical systems because a bug in this computation may cause both sensors to fail.

The second dice, which yields code unique to A, is helpful in locating bugs in computation on A if computation of B is correct. For example, consider a program that counts the number of words and characters in a file. If the final value of `character_count` is correct, but the final value of `word_count` is not then the second kind of dice could be applied. In this case, it contains statements that affect the value of `word_count` but not the value of `character_count`. This implies the looping and reading of characters from the file need not be considered.

The original work on dicing considered only backward slices. Incorporating forward slices increases the usefulness of dicing. For example, the notion of *program chopping* identifies the statement that transmit values from a statement t to a statement s [49]: $chop(t, s)$ includes those program points affected by the computation at program point t that affect the computation at program point s. A program chop is useful in debugging when a change at t causes an incorrect result to be produced at s. The statements in $chop(t, s)$ are the statements that transmit the effect of the change at t to s. Debugging attention should be focused there. In the absence of procedures, $chop(t, s)$ is simply the intersection of the forward slice taken with respect to t and the backward slice taken with respect to s and can be viewed as a generalized kind of program dice. The same is not true for interprocedural chopping [85].

As initially introduced, s and t must be in the same procedure, P, and only components from P are reported. This was later generalized to allow s and t to be in different procedures and to contain components for procedure other than P [85]. It is interesting to note that for interprocedural chopping the dicing idea of intersections a forward and backward slice is imprecise. Interprocedural version of other set theoretic operations on slices that work with intraprocedural slices, have eluded researchers [81]. Precise interprocedural chopping is addressed in [85].

3.6 Software Quality Assurance

Software quality assurance auditors are faced with a myriad of difficulties, ranging from inadequate time to inadequate computer-aided software engineering (CASE) tools [33]. One particular problem is the location of safety critical code that may be interleaved throughout the entire system. Moreover, once this code is located, its effects throughout the system are difficult to ascertain. Program slicing is applied to mitigate these difficulties in two ways. First, program slicing can be used to locate all code that contributes to the value of variables that might be part of a safety critical component. Second, slicing-based techniques can be used to validate *functional diversity* (*i.e.*, that there are no interactions of one safety critical component with another safety critical component and that there are no interactions of non safety critical components with the safety critical components).

A design error in hardware or software, or an implementation error in software may result in a *Common Mode Failure* of redundant equipment. A common mode failure is a failure as a result of a common cause, such as the failure of a system caused by the incorrect computation of an algorithm. For example, suppose that **X** and **Y** are distinct critical outputs and that **X** measures a rate of increase while **Y** measures a rate of decrease. If the computation of *both* of the rates depends on a call to a common numerical differentiator, then a failure in the differentiator can cause a common mode failure of **X** and **Y**.

One technique to defending against common mode failures uses *functional diversity.* Functional diversity in design is a method of addressing the common mode failure problem in software that uses multiple algorithms on independent inputs. Functional diversity allows the same function to be executed along two or more independent paths.

One technique to solve this problem combines *Fault Tree Analysis* and program slicing. Once the system hazards have been identified, the objective of fault tree analysis is to mitigate the risk that they will occur. One approach to achieving this objective is to use *system fault tree analysis.* Under the assumption that there are relatively few unacceptable system states and that each of these hazards has been determined, the analysis procedure is as follows. The auditor assumes that a hazard has occurred and constructs a tree with the hazardous condition as the root. The next level of the tree is an enumeration of all the necessary preconditions for the hazard to occur. These conditions are combined with logical *and* and *or* as appropriate. Then each new node is expanded "similarly until all leaves have calculable probability or cannot be expanded for some reason"[59].

System fault tree analysis gives the auditor the sub-components of the system that must be carefully examined. Part of this examination is the validation that there are no interactions with non-critical functions. The determination of the specific components that will be examined is up to the auditor. This information should be obtainable from the design documentation.

Slicing is used as an aid to validating safety as follows. First, the auditor uses system fault tree analysis to locate critical components. The software that is invoked when a hazardous condition occurs is identified in the system. The auditor then locates the software variables that are the indicators of unsafe conditions. Program slices are extracted on these "interesting" variables. These slices can be used to validate that there are no interactions between critical components or with non-critical components using program dicing.

Program slices can also be used to assure diversity: computed from the outputs of individual hazards can be examined to determine the logical independence of the events. For instance, if A and B are two critical conditions, the dice computed by intersecting the program slices on these two conditions provides partial information on whether or not both conditions can occur simultaneously. If the dice is empty then there is no way that the software can guarantee that both will not occur simultaneously (there may be other ways the verify that both will not occur). If the dice in not empty, inspection of the overlap *may* prove that both conditions cannot occur together (although the functional diversity of such computations is suspect).

These program projections can also be highlighted for more vigorous analysis, inspection and testing. A static program slice can be further refined by examining the trajectory of specific inputs through the program; dynamic slices are used to observe individual instances of the computation. This simplifies the tedious task of the auditor and permits undivided attention to be focused on the analytic portions of the audit.

The utility of a slicing tool comes from automating the task of finding statements that are relevant to a computation. Without any tool, the software quality assurance auditor evaluating functional diversity would examine the program under consideration until outputs were identified that should be computed independently. The auditor would then try to verify independence by reading code.

Unravel [67] is a static program slicer developed at the National Institute of Standards and Technology as part of a research project. It slices ANSI-C programs. Its only limitations are in the treatment of **unions**, **forks**, and pointers to functions. The tool is divided into three main components: a source code analysis component to collect information necessary for the computation of program slices, a link component to link flow information from separate source files together and an interactive slicing component that the software quality assurance auditor can use to extract program components and statements for answering questions about the software being audited.

3.7 Reverse Engineering

Reverse engineering concerns the problem of comprehending the current design of a program and the way this design differs from the original design. This involves abstracting out of the source code the design decisions and rationale from the initial development (design recognition) and understanding the algorithms chosen (algorithm recognition).

Program slicing provides a toolset for this type of re-abstraction. For example, a program can be displayed as a lattice of slices ordered by the *is-a-slice-of* relation [32, 81, 12]. Comparing the original lattice and the lattice after (years of) maintenance can guide an engineer towards places where reverse engineering energy should be spent. Because slices are not necessarily contiguous blocks of code they are well suited for identifying differences in algorithms that may be span multiple blocks or procedures.

Beck and Eichmann observer that elements "towards the bottom" of this lattice are often *clichés* [12]. For example, in the word count program the slice that reads the characters from a file is contained in (is-a-slice-of) three other slices (these slices count the number of words, lines, and characters in the input). The common slice is the read-a-file clichè.

Beck and Eichmann also propose the notion of interface slicing for use in reverse engineering. Understanding a program often requires identifying its major abstractions and there interfaces. An interface slice is essentially a forward slice taken with respect to the entry vertices in a collection of procedures [12]. This projection of a general software module (*e.g.*, a set, list, or window widget), captures the particular behaviors required for a particular use.

An interface slice is computed from an interface dependence graph as a forward graph traversal. The dual of this operation uses a backward graph traversal (*i.e.*, traverses the dependence edges from target to source). Starting from all calls on procedure P, this "backward" interface slice includes the public interfaces for those procedures (from other modules) that require P.

While interface slicing is useful in reverse engineering, it seems more useful in reshaping the development process. In particular, as Beck and Eichmann observe, a programmer with access to a large repository of software modules often wants only part of the functionality of a module in the repository. Presently, the programmer has two unsatisfactory options: (1) create a special copy of the module, or (2) include unwanted code. The first option requires access to the source code, which may not be possible. It also creates multiple copies of some functions from the module, which complicates later updates. The second option increases the code size and may degrade the performance of the compiler when optimizing the code.

Interface slicing can be used to provide a third alternative that has neither of these deficiencies: the complete module is made available to the interface slicer. A programmer, desiring partial functionality from a module, tells the interface slicer which exported functions are required. The interface slicer the produces the public interface for the required functions without releasing the source for their implementation. Thus a specialized version of the original is made available to the programmer without introducing a new copy or releasing proprietary source code.

3.8 Functional Cohesion

Cohesion is an attribute of a software unit that purports to measure the "relatedness" of the unit. Cohesion has been qualitatively characterized as *coincidental, logical, procedural, communicational, sequential* and *functional*, with coincidental being the weakest and functional being the strongest[98]. Yourdon and Constantine note that functional cohesion "is an integral part of, and is essential to, the performance of a single function."[98]

To construct a slicing-based measure of functional cohesion, Bieman and Ott [16] define *data slices*, a backward and forward static slice that uses data tokens (variable and constant definitions and references) rather than statements as the unit of decomposition. This definition ensures that any change will impact at least one slice and leads to a *slice abstraction* model of the procedure under consideration: one can regard the slice as the sequence of variable tokens that are contained in it.

The tokens that are in every data slice are referred to as *super-glue*; tokens that are in more than 1 data slice are referred to as *glue*. The metrics are based on the ratios of the appearance of glue and super-glue tokens in a slice. *Strong Functional Cohesion* is the ratio of super-glue tokens in the slice to the number of tokens in the slice. *Weak Functional Cohesion* is the ratio of glue tokens in the slice to the number of tokens in the slice.

Another method for measuring cohesion is to measure the *adhesiveness* of the individual tokens. A token that glues five data slices together is more adhesive than a token that glues only two data slices together. Thus, the adhesion of an individual token is the ratio of number of slices in which the token appears to the number of data slices in a procedure.

Bieman and Ott show that these metrics form a well-defined, ordinal measure of the functional cohesion of a program. That is, they show the orderings imposed match ones intuition. They are unable to show that are on a ratio scale (that meaningful composition of the metrics exist). The measures are not additive. The study of the relationship of these metrics to product attributes such as reliability and maintainability is ongoing.

References

[1] H. Agrawal. Towards automatic debugging of computer programs. Technical Report SERC-TR-40-P, Purdue University, 1989.

[2] H. Agrawal. On slicing program with jump statements. In *Proceedings of the ACM SIGPLAN 94 Conference on Programming Language Design and Implementation*, pages 303–312, June year.

[3] H. Agrawal, R. DeMillo, and E. Spafford. A process state model to relate testing and debugging. Technical Report SERC-TR-27-P, Purdue University, 1988.

[4] H. Agrawal and R.A. DeMIllo. Dynamic slicing in the presence of unconstrained pointers. In *Proceedings of the ACM Symposium on Testing and Verification*, October 1991.

[5] H. Agrawal and J. Horgan. Dynamic program slicing. Technical Report SERC-TR-56-P, Purdue University, 1989.

[6] H. Agrawal and J. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN '90 Conference*, 1990.

[7] T . Ball and S. Horwitz. Slicing programs with arbitrary control-flow. In *Proceedings of the 1st International Workshop on Automated and Algorithmic Debugging, (appears in Lecture Notes in Computer Science 749)*, pages 206–222, 1993.

[8] J.P. Banning. 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)*, 1979.

[9] Barendergt, Eekelen, Glauert, Kennawar, and Plasneijer. Term graph rewriting. In *In proceedings of the PARLE Conference, Vol II: Parallel Languages*, 1987.

[10] J. M. Barth. A practical interprocedural dataflow analysis algorithm. *Communications of the Association for Computing Machinery*, 21(9):724–726, September 1978.

[11] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages,*. ACM, 1993.

[12] J. Beck. Program and interface slicing for reverse engineering. In *Proceeding of the Fifteenth International Conference on Software Engineering*, 1993. also in Proceedings of the Working Conference on Reverse Engineering.

[13] J-F. Bergeretti and B. Carré. Information-flow and data-flow analysis of **while**-programs. *ACM Transactions on Programming Languages and Systems*, 7(1):37–61, January 1985.

[14] V. Berzins. On merging software extensions. *Acta Informatica*, 23:607–619, 1985.

[15] V. Berzins. Software merge: Models and methods for combining changes to programs. *International Journal on Systems Integration*, 1:121–141, August 1991.

[16] J. Bieman and L. Ott. Measuring functional cohesion. *IEEE Transactions on Software Engineering*, 20(8):644–657, August 1994.

[17] D. Binkley. Using semantic differencing to reduce the cost of regression testing. In *Proceedings of the Conference on Software Maintenance - 1992*, pages 41–50, November 1992.

[18] D. Binkley. Slicing in the presence of parameter aliasing. In *Software Engineering Research Forum*, Orlando, FL, November 1993.

[19] D. Binkley. Precise executable interprocedural slices. *ACM Letters on Programming Languages and Systems*, 1-4(2), 1994.

[20] D. Binkley. Reducing the cost of regression testing by semantics guided test case selection. In *IEEE International Conference on Software Maintenance*, 1995.

[21] D. Binkley, S. Horwitz, and T. Reps. Prograom integration for languages with procedure calls. *ACM Transactions on Software Engineering and Methodology*, 4(1):3–35, January 1995.

[22] J. Choi and J. Ferrante. Static slicing in the presence of GOTO statements. *ACM Transacton on Programming Languages and Systems*, 16(4):1097–1113, July 1994.

[23] J-D. Choi, B. Miller, and P. Netzer. Techniques for debugging parallel programs with flowback analysis. Technical Report 786, University of Wisconsin - Madison, August 1988.

[24] F. Cutillo, R. Fiore, and G. Visaggio. Identification and extraction of domain independent components in large programs. In *Proceedings of the Working Conference on Reverse Engineering*, pages 83–92, June 1993.

[25] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the Association for Computing Machinery*, 20(7):504–513, July 1977.

[26] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[27] J. Field, G . Ramalingam, and F. Tip. Parametric program slicing. In *Conference Record of the Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 379–392, 1995.

[28] K. Gallagher. The surgeon's assistant. In *Software Engineering Research Forum*, Boca Raton, FL, November 1995.

[29] K. B. Gallagher. Surgeon's assistant limits side effects. *IEEE Software*, May 1990.

[30] K. B. Gallagher. Conditions to assure semantically correct consistent software merges in linear time. In *Proceedings of the Third International Workshop on Software Configuration Management*, pages 80–84, May 1991.

[31] K. B. Gallagher. Using program slicing to eliminate the need for regression testing. In *Eighth International Conference on Testing Computer Software*, May 1991.

[32] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, August 1991.

[33] K. B. Gallagher and J. R. Lyle. Program slicing and software safety. In *Proceedings of the Eighth Annual Conference on Computer Assurance*, pages 71–80, June 1993. COMPASS '93.

[34] M. Gandle, A. Santal, and G. Venkatesh. Slicing functional programs using collecting abstract interpretation. In *First Symposium on Algorithmic and Automated Debugging, Linkoping, Sweeden*, 1993. Short Presentation.

[35] R. Gopal. Dynamic program slicing based on dependence relations. In *Proceedings of the IEEE Conference on Software Maintenance*, pages 191–200, 1991.

[36] R. Gupta, M.J. Harrold, and M.L. Soffa. An approach to regression testing using slicing. In *Proceedings of the IEEE Conference on Software Maintenance*, pages 299–308, 1992.

[37] M. Harman, S. Danicic, and Y. Sivaguranathan. A parallel algorithm for static program slicing. *Information Processing Letters*, 1996.

[38] M.J. Harrold, B.Malloy, and G. Rothermel. Efficient construction of program dependence graphs. In *International Symposium on Software Testing and Analysis (ISSTA '93)*, 1993.

[39] P. Hausler. Denotational program slicing. In *Proceedings of the 22nd Hawaii International Conference on System Sciences*, pages 486–494, January 1989. Volume II, Software Track.

[40] S. Horwitz, J. Prins, and T. Reps. Integrating non-interfering versions of programs. In *Proceedings of the SIGPLAN 88 Symposium on the Principles of Programming Languages*, January 1988.

[41] S. Horwitz, J. Prins, and T. Reps. On the adequacy of program dependence graphs for representing programs. In *Proceedings of the SIGPLAN 88 Symposium on the Principles of Programming Languages*, January 1988.

[42] S. Horwitz, J. Prins, and T. Reps. Support for integrating program variants in an environment for programming in the large. In *Proceedings of the International Workshop on Software Version and Configuration Control 88*, Grassau, Germany, January 1988.

[43] S. Horwitz, J. Prins, and T. Reps. Integrating non-interfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, July 1989.

[44] S. Horwitz and T. Reps. Efficient comparison of program slices. Technical Report 983, University of Wisconsin at Madison, 1990.

[45] S. Horwitz and T. Reps. Efficient comparison of program slices. *Acta Informatica*, pages 713–732, 1991.

[46] S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In *Proceedings of the Fourteenth International Conference on Software Engineering*, 1992.

[47] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 88 Conference on Programming Language Design and Implementation*, Atlanta, Georgia, June 1988.

[48] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):35–46, January 1990.

[49] D. Jackson and E.J. Rollins. A new model of program dependences for reverse engineering. In *Proceedings of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering* (New Orleans, LA), December 1994.

[50] M. Kamkar. *Interprocedural Dynamic Slicing with Applications to Debugging and Testing.* PhD thesis, Linkoping University, S-581 83 Linkoping, Sweeden, 1993.

[51] M. Kamkar, P. Fritzson, and N. Shahmehri. Interprocedural dynamic slicing applied to interprocedural data flow testing. In *Proceeding of the Conference on Software Maintenance -93*, pages 386–395, 1993.

[52] K. Kennedy. A survey of data flow analysis techniques. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[53] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, October 1988.

[54] B. Korel and J. Laski. STAD - A system for testing and debugging: User perspective. In *Proceedings of the Second Workshop on Software Testing, Verification and Analysis*, pages 13–20, Banff, Alberta, Canada, July 1988.

[55] B. Korel and J. Laski. Dynamic slicing of computer programs. *Journal of Systems and Software*, pages 198–195, 1990.

[56] F. Lanubile and G. Visaggio. Function recovery based on program slicing. In *Proceedings of the Conference on Software Maintenance - 1993*, pages 396–404, September 1993.

[57] J. Laski. Data flow testing in STAD. *The Journal of Systems and Software*, 1989.

[58] J. Laski and W. Szermer. Identification of program modifications and its application in software maintenance. In *Proceedings of the Conference on Software Maintenance - 1992*, pages 282–290, November 1992.

[59] N. Leveson, S. Cha, and T. Shimeall. Safety verification of Ada programs using software fault trees. *IEEE Computer*, 8(4):48–59, May 1991.

[60] R. Linger, H. Mills, and B. Witt. *Structured Programming: Theory and Practice*. Addison-Wesley, Reading, Massachusetts, 1979.

[61] P. Livadas and S. Croll. A new method in calculating transitive dependences. *Journal of Software Maintenance*, 1994.

[62] H. Longworth. Slice based program metrics. Master's thesis, Michigan Technological University, Houghton, Michigan, 1985.

[63] H. Longworth, L. Ott, and M.Smith. The relationship between program complexity and slice complexity during debugging tasks. In COMPSAC *86*, 1986.

[64] J. R. Lyle. *Evaluating Variations of Program Slicing for Debugging*. PhD thesis, University of Maryland, College Park, Maryland, December 1984.

[65] J. R. Lyle and M. D. Weiser. Experiments on slicing-based debugging aids. In Elliot Soloway and Sitharama Iyengar, editors, *Empirical Studies of Programmers*. Ablex Publishing Corporation, Norwood, New Jersey, 1986.

[66] J. R. Lyle and M. D. Weiser. Automatic program bug location by program slicing. In *Proceeding of the Second International Conference on Computers and Applications*, pages 877–882, Peking, China, June 1987.

[67] J.R. Lyle, D.R. Wallace, J.R. Graham, K.B. Gallagher, J.E. Poole, and D.W. Binkley. *A CASE tool to evaluate functional diversity in high integrity software*. U.S. Department of Commerce, Technology Administration, National Institute of Standards and Technology, Computer Systems Laboratory, Gaithersburg, MD, 1995.

[68] D. Maydan, J. Hennessy, and M. Lam. Efficient and exact data dependence analysis. In *Proceedings of the ACM SIGPLAN 91 Conference on Programming Language Design and Implementation, Toronto, Ontario, June, 1991, ACM SIGPLAN Notices*, 1991.

[69] E. Merlo, J. Girard, L. Hendren, and P. De Mori. Multi-valued constant propagation for the reengineering of user interfaces. In *Proceedings of the Conference on Software Maintenance - 1993*, pages 120–129, September 1993.

[70] K. Ono, K. Maruyama, and Y. Fukazawa. Applying a verification method and a decomposition method to program modification. *Trans. IEICE*, J77-D-I(11), November 1994. JAPAN.

[71] L. Ott and J. Bieman. Effects of software changes on module cohesion. In *Proceedings of the Conference on Software Maintenance - 1992*, pages 345–353, November 1992.

[72] L. Ott and J. Thuss. The relationship between slices and module cohesion. In *International Conference on Software Engineering*, May 1989.

[73] K. Ottenstein and L. Ottenstein. The program dependence graph in software development environments. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184, May 1984. L. Ottenstein is now known as L. Ott.

[74] M. Platoff and M. Wagner. An integrated program representation and toolkit of the maintenance of C programs. In *Proceeding of the Conference on Software Maintenance*, October, 1991.

[75] M. Pleszcoch, P. Hausler, A. Hevner, and R. Linger. Function theoretic principles of program understanding. In *Proceedings of the Twenty-third Annual Hawaii Conference on System Sciences*, volume 4, pages 74–81, 1990.

[76] W. Pugh and D. Wonnacott. Eliminating false data dependences using the omega test. In *Proceedings of the ACM SIGPLAN 92 Conference on Programming Language Design and Implementation*, pages 140–151, 1992.

[77] G. Ramalingam and T. Reps. A theory of program modifications. In S. Abramsky and T.S.E. Maibaum, editors, *Proceedings of the Colloquium on Combining Paradigms for Software Development*, pages 137–152. Springer-Verlag, 1991.

[78] G. Ramalingam and T. Reps. Modification algebras. In *Proceedings of the Second International Conference on Algebraic Methodology and Software Technology*. Springer-Verlag, 1992.

[79] S. Rapps and E.J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, 1985.

[80] T. Reps. On the algebraic properties of program integration. Technical Report 856, University of Wisconsin at Madison, June 1989.

[81] T. Reps. Algebraic properties of program integration. *Science of Computer Programming*, 17:139–215, 1991.

[82] T. Reps and T. Bricker. Semantics-based program integration illustrating interference in interfering versions of programs. In *Proceedings of the Second International Workshop on Software Configuration Management*, pages 46–55, Princeton, New Jersey, October 1989.

[83] T. Reps and S. Horwitz. Semantics-based program integration. In *Proceedings of the Second European Symposium on Programming (ESOP '88)*, pages 133–145, Nancy, France, March 1988.

[84] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. Speeding up slicing. In *Proceedings of the Second ACM SIGSOFT Symposium on Foudations of Software Engineering*, pages 11–20, 1994. Published in ACM SIGSOFT Notes v19 n4.

[85] T. Reps and G. Rosay. Precise interprocedural chopping. In *Proceedings of the third ACM Symposium on the Foundations of Software Engineering* (Washington, DC), October 1995.

[86] T. Reps and W. Yang. The semantics of program slicing. Technical Report 777, University of Wisconsin - Madison, June 1988.

[87] G. Rothermel and M.J. Harrold. Selecting tests and identifying test coverage requirements for modified software. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 169–84, August 1994.

[88] T. Shimomura. The program slicing technique and its application to testing, debugging and maintenance. *Journal of IPS of Japan*, 9(9):1078–1086, September 1992.

[89] Frank Tip. *Generation of Program Analysis Tools*. PhD thesis, University of Amsterdam, 1995.

[90] G. Venkatesh. Experimental results from dynamic slicing of c programs. *ACM Transactions on Programming Languages and Systems*, 17(2):197–216, March 1995.

[91] G.A. Venkatesh. The semantic approach to program slicing. In *Proceedings of the ACM SIGPLAN 91 Conference on Programming Language Design and Implementation, Toronto, Ontario*, June 26-28 1991.

[92] D. Weise, R. Crew, M. Ernst, and B. Steensgaard. Value depedence graphs: Representation without taxation. In *Proceedings of the ACM SIGPLAN-SIGACT Twentifirst Symposium on Principles of Programming Languages*, pages 297–310, January 1994.

[93] M. Weiser. *Program Slicing: Formal, Psychological and Practical Investigations of an Automatic Program Abstraction Method.* PhD thesis, The University of Michigan, Ann Arbor , Michigan, 1979.

[94] M. Weiser. Program slicing. In *Proceeding of the Fifth International Conference on Software Engineering*, pages 439–449, May 1981.

[95] M. Weiser. Programmers use slices when debugging. *CACM*, 25(7):446–452, July 1982.

[96] M. Weiser. Reconstructing sequential behavior from parallel behavior projections. *Information Processing Letters*, 17(5):129–135, October 1983.

[97] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10:352–357, July 1984.

[98] E. Yourdon and L. Constantine. *Structured Design.* Prentice-Hall, Englewood Cliffs, New Jersey, 1979.