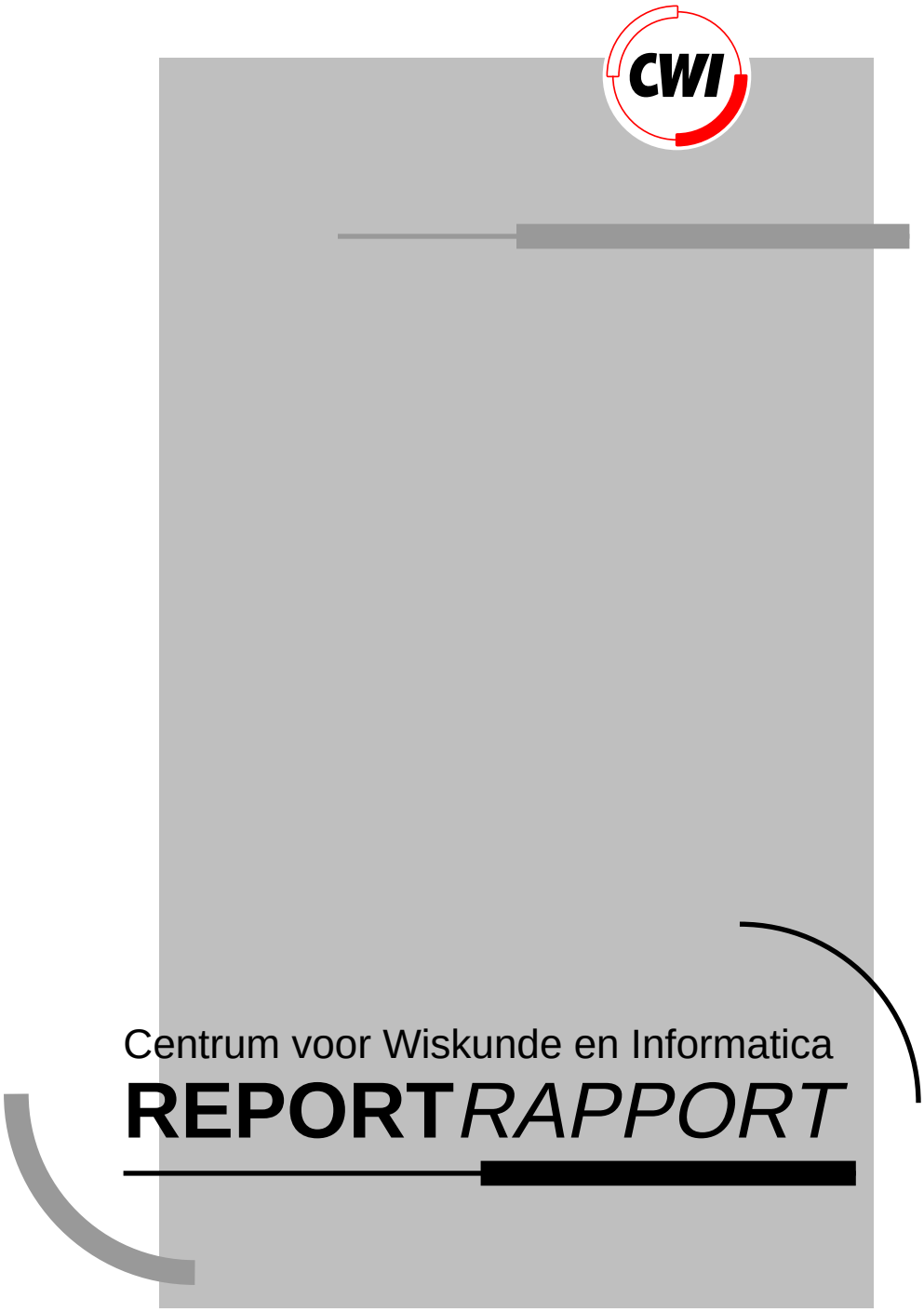


Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

A Survey of Program Slicing Techniques

F. Tip

Computer Science/Department of Software Technology

CS-R9438 1994

A Survey of Program Slicing Techniques

Frank Tip

CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

tip@cwi.nl

Abstract

A *program slice* consists of the parts of a program that (potentially) affect the values computed at some point of interest, referred to as a *slicing criterion*. The task of computing program slices is called *program slicing*. The original definition of a program slice was presented by Weiser in 1979. Since then, various slightly different notions of program slices have been proposed, as well as a number of methods to compute them. An important distinction is that between a *static* and a *dynamic* slice. The former notion is computed without making assumptions regarding a program's input, whereas the latter relies on some specific test case.

Procedures, arbitrary control flow, composite datatypes and pointers, and interprocess communication each require a specific solution. We classify static and dynamic slicing methods for each of these features, and compare their accuracy and efficiency. Moreover, the possibilities for combining solutions for different features are investigated. We discuss how compiler-optimization techniques can be used to obtain more accurate slices. The paper is concluded with an overview of the applications of program slicing, which include debugging, program integration, dataflow testing, and software maintenance.

1991 Mathematics Subject Classification: 68Q55 [**Theory of computing**]: Semantics, 68Q60 [**Theory of computing**]: Specification and verification of programs.

1991 CR Categories: D.2.2 [**Software engineering**]: Tools and Techniques, D.2.5 [**Software engineering**]: Testing and debugging, D.2.6 [**Software engineering**]: Programming environments, D.2.7 [**Software engineering**]: Distribution and Maintenance.

Key Words & Phrases: Program slicing, static slicing, dynamic slicing, program analysis, debugging, data dependence, control dependence, program dependence graph.

> [1] **slice** \'slīs\ *n* **1 :** a thin flat piece cut from something **2 :** a wedge-shaped blade (as for serving fish) **3 :** a flight of a ball (as in golf) that curves in the direction of the dominant hand of the player propelling it
> [2] **slice** *vb* **sliced**; **slic-ing 1 :** to cut a slice from; *also* to cut into slices **2 :** to hit (a ball) so that a slice results
>
> *The Merriam-Webster Dictionary*

1 Overview

We present a survey of algorithms for program slicing that can be found in the present literature. A *program slice* consists of the parts of a program that (potentially) affect the values computed at some point of interest, referred to as a *slicing criterion*. Typically, a slicing criterion consists of a pair (line-number, variable). The parts of a program which have a direct or indirect effect on the values computed at a slicing criterion C are called the *program slice with respect to criterion* C. The task of computing program slices is called *program slicing*.

The original concept of a program slice was introduced by Weiser [82, 83, 85]. Weiser claims that a slice corresponds to the mental abstractions that people make when they are debugging a program, and suggests the integration of program slicers in debugging environments. Various slightly different notions of program slices have been proposed, as well as a number of methods to compute slices. The main reason for this diversity is the fact that different applications require different properties of slices. Weiser defined a program slice S as a *reduced, executable program* obtained from a program P by removing statements, such that S replicates part of the behavior of P. Another common definition of a slice is a *subset* of the statements and control predicates of the program which directly or indirectly affect the values computed at the criterion, but which do not necessarily constitute an executable program. An important distinction is that between a *static* and a *dynamic* slice. The former notion is computed without making assumptions regarding a program's input, whereas the latter relies on some specific test case. Below, in Sections 1.1 and 1.2, we consider these notions in some detail.

Features of programming languages such as procedures, arbitrary control flow, composite datatypes and pointers, and interprocess communication each require a specific solution. Static and dynamic slicing methods for each of these features are classified and compared in terms of accuracy and efficiency. In addition, we investigate the possibilities for integrating solutions for different language features. Throughout this paper, slicing algorithms are compared by applying them to similar examples.

1.1 Static Slicing

Figure 1 **(a)** shows an example program which asks for a number `n`, and computes the sum and the product of the first `n` positive numbers. Figure 1 **(b)** shows a slice of this program with respect to criterion (10, `product`). As can be seen in the figure, all computations involving variable `sum` have been 'sliced away'.

In Weiser's approach, slices are computed by computing consecutive sets of indirectly relevant statements, according to data flow and control flow dependences. Only statically available information is used for computing slices; hence, this type of slice is referred to as a *static* slice. An alternative method for computing static slices was suggested by Ottenstein and Ottenstein [69], who restate the problem of static slicing in terms of a reachability problem in a *program dependence graph* (PDG) [27, 58]. A PDG is a directed graph with vertices corresponding to statements and control predicates, and edges corresponding to data and control dependences. The slicing criterion is identified with a vertex in the PDG, and a slice corresponds to all PDG vertices from which the vertex under consideration can be reached. Various program slicing approaches we discuss later utilize modified and extended versions of PDGs as their underlying program representation. Yet

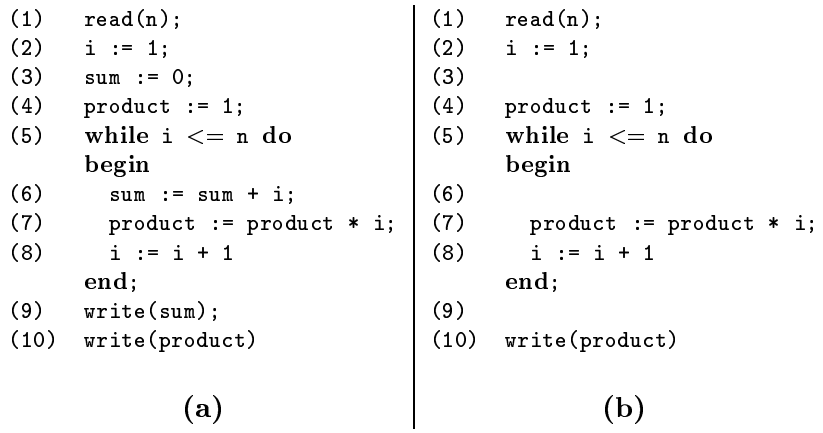


Figure 1: **(a)** An example program. **(b)** A slice of the program w.r.t. criterion (10, `product`).

another approach was proposed by Bergeretti and Carré [16], who define slices in terms of information-flow relations which are derived from a program in a syntax-directed fashion.

The slices mentioned so far are computed by gathering statements and control predicates by way of a *backward* traversal of the program, starting at the slicing criterion. Therefore, these slices are referred to as *backward* (static) slices. Bergeretti & Carré were the first to define a notion of a *forward* static slice in [16], although Reps and Bricker were the first to use this terminology [73]. Informally, a forward slice consists of all statements and control predicates dependent on the slicing criterion, a statement being 'dependent' on the slicing criterion if the values computed at that statement depend on the values computed at the slicing criterion, or if the values computed at the slicing criterion determine the fact if the statement under consideration is executed or not. Backward and forward slices[1] are computed in a similar way; the latter requires tracing dependences in the forward direction.

1.2 Dynamic Slicing

Although the exact terminology 'dynamic program slicing' was first introduced by Korel and Laski in [56], dynamic slicing may very well be regarded as a non-interactive variation of Balzer's notion of flowback analysis [10]. In flowback analysis, one is interested how information flows through a program to obtain a particular value: the user interactively traverses a graph that represents the data and control dependences between statements in the program. For example, if the value computed at statement s depends on the values computed at statement t, the user may trace back from the vertex corresponding to statement s to the vertex for statement t. Recently, flowback analysis has been implemented efficiently for parallel programs [22, 67].

In the case of dynamic program slicing, only the dependences that occur in a *specific* execution of the program are taken into account. A *dynamic slicing criterion* specifies the input, and distinguishes between different occurrences of a statement in the execution history; typically, it consists of triple (input, occurrence of a statement, variable). An alternate view of the difference between static and dynamic slicing is that dynamic slicing

[1] Unless stated otherwise, "slice" will denote "backward slice" in the sequel.

```
(1)   read(n);
(2)   i := 1;
(3)   while (i <= n) do
      begin
(4)     if (i mod 2 = 0) then
(5)       x := 17
        else
(6)       x := 18;
(7)     i := i + 1
      end;
(8)   write(x)
```

(a)

```
(1)   read(n);
(2)   i := 1;
(3)   while (i <= n) do
      begin
(4)     if (i mod 2 = 0) then
(5)       x := 17
        else
(6)             ;
(7)     i := i + 1
      end;
(8)   write(x)
```

(b)

Figure 2: **(a)** Another example program. **(b)** Dynamic slice w.r.t. criterion (`n = 2`, 8^1, `x`).

assumes a *fixed* input for a program, whereas static slicing does not make assumptions regarding the input. Hybrid approaches, where a combination of static and dynamic information is used to compute slices, are described in [22, 26, 49, 79].

Figure 2 shows an example program and its dynamic slice w.r.t. the criterion (`n = 2`, 8^1, `x`), where 8^1 denotes the first occurrence of statement 8 in the execution history of the program. Note that for input `n = 2`, the loop is executed twice, and that the assignments `x := 17` and `x = 18` are each executed once. In this example, the **else** branch of the **if** statement may be omitted from the dynamic slice since the assignment of 18 to variable `x` in the first iteration of the loop is 'killed' by the assignment of 17 to `x` in the second iteration[2]. By contrast, the *static* slice of the program in Figure 2 **(a)** w.r.t. criterion (8, `x`) consists of the entire program.

1.3 Earlier Work

There are a number of earlier frameworks for comparing slicing methods, as well as some earlier surveys of slicing methods.

In [79], Venkatesh presents formal definitions of several types of slices in terms of denotational semantics. He distinguishes three independent dimensions according to which slices can be categorized: static vs. dynamic, backward vs. forward, and closure vs. executable. Some of the slicing methods in the literature are classified according to these criteria [5, 41, 44, 57, 69, 85]. Moreover, Venkatesh introduces the concept of a *quasi-static* slice. This corresponds to situations where some of the inputs of a program are fixed, and some are unknown. No constructive algorithms for computing slices are presented in [79].

In [59], Lakhotia restates a number of static slicing methods [41, 69, 85] as well as the program integration algorithm of [41] in terms of operations on directed graphs. He presents a uniform framework of *graph slicing*, and distinguishes between *syntactic* properties of slices which can be obtained solely through graph-theoretic reasoning, and *semantic* properties which involve interpretation of the graph representation of a slice. Although the paper only addresses static slicing methods, it is stated that the dynamic slicing methods of [5, 57] may be modeled in a similar way.

[2] In fact, one might argue that the **while** construct may be replaced by the **if** statement in its body. This type of slice will be discussed in Section 5.

Gupta and Soffa present a generic algorithm for *static slicing* and the solution of related dataflow problems (such as determining reaching definitions) that is based on performing a traversal of the control flow graph (CFG) [35]. The algorithm is parameterized with: (i) the *direction* in which the CFG should be traversed (backward or forward), (ii) the *type* of dependences under consideration (data and/or control dependence), (iii) the *extent* of the search (i.e., should only immediate dependences be taken into account, or transitive dependences as well), and (iv) whether only the dependences that occur along *all* CFG-paths paths, or dependences which occur along *some* CFG-path should be taken into account. A slicing criterion is either a *set of variables* at a certain program point or a *set of statements*. For slices that take data dependences into account, one may choose between the values of variables *before* or *after* a statement.

In [43], Horwitz and Reps present a survey of the work that has been done at the University of Wisconsin-Madison on slicing, differencing and integration of single-procedure and multi-procedure programs, as operations on program dependence graphs. In addition to discussing the motivation for this work in considerable detail, the most significant definitions, algorithms, theorems, and complexity results that can be found in [37, 39, 41, 42, 44, 76] are presented.

An earlier classification of static and dynamic slicing methods was presented by Kamkar in [48, 49]. The differences between Kamkar's work and ours may be summarized as follows. First, our paper is more up-to-date and more complete; for instance, Kamkar does not address any of the papers that discuss slicing in the presence of arbitrary control flow [2, 8, 9, 21] or methods for computing slices that are based on information-flow relations [16, 33]. Second, the papers are organized in a different way. Whereas Kamkar discusses each slicing method and its applications separately, this paper is organized in terms of a number of 'orthogonal' problems, such as the problems posed by procedures, or composite variables, aliasing, and pointers. This approach enables us to address the possibilities for combining solutions to different 'orthogonal' problems. Third, unlike Kamkar's work we compare the accuracy and efficiency of slicing methods, and we attempt to determine the fundamental strengths and weaknesses of each slicing method, irrespective of its original presentation. Finally, we suggest a number of directions for improving the accuracy of slicing algorithms.

1.4 Organization of the Paper

The remainder of this paper is organized as follows. In Section 2, we will introduce the cornerstones of most slicing algorithms: the notions of data dependence and control dependence. Readers familiar with these concepts may skip this section and consult it on demand. Section 3 contains an overview of static slicing methods. First, the simple case of slicing structured programs with scalar variables only is studied. Then, we address algorithms for slicing in the presence of procedures, arbitrary control flow, composite variables and pointers, and interprocess communication. Section 3.6 compares and classifies methods for static slicing. Section 4 addresses dynamic slicing methods, and is organized in a similar way as Section 3. Section 5 suggests how compiler-optimization techniques may be used to obtain more accurate slices. Applications of program slicing are discussed in Section 6. Finally, Section 7 summarizes the main conclusions of this survey.

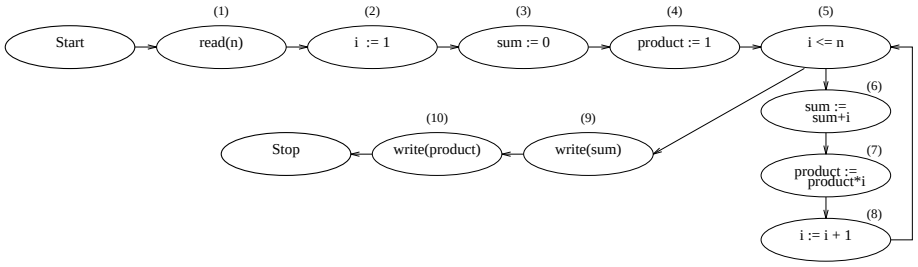


Figure 3: CFG of the example program of Figure 1 **(a)**.

2 Data Dependence and Control Dependence

Data dependence and control dependence are defined in terms of the CFG of a program. A CFG contains a node for each statement and control predicate in the program; an edge from node i to node j indicates the possible flow of control from the former to the latter. CFGs contain special nodes labeled START and STOP corresponding to the beginning and the end of the program, respectively.

The sets DEF(i) and REF(i) denote the sets of variables defined and referenced at CFG node i, respectively. Several types of data dependences can be distinguished, such as flow dependence, output dependence and anti dependence [27]. Flow dependences can be further classified as being loop-carried or loop-independent, depending whether or not they arise as a result of loop iteration. For the purposes of slicing, only flow dependence is relevant, and the distinction between loop-carried and loop-independent flow dependences can be ignored. Node j is *flow dependent* on node i if there exists a variable x such that:

- $x \in$ DEF(i),
- $x \in$ REF(j), and
- there exists a path from i to j without intervening definitions of x.

Alternatively stated, the definition of x at node i is a *reaching definition* for node j.

Control dependence is usually defined in terms of post-dominance. A node i in the CFG is *post-dominated* by a node by j if all paths from i to STOP pass through j. A node j is *control dependent* on a node i if:

- there exists a path P from i to j such that any $u \neq i$, j in P is post-dominated by j, and
- i is not post-dominated by j.

Determining the control dependences in programs with arbitrary control flow is studied in [27]. For programs with structured control flow, control dependences can be determined in a simple syntax-directed manner [40]: the statements in the branches of an **if** or **while** are control dependent on the control predicate.

As an example, Figure 3 shows the CFG for the example program of Figure 1 **(a)**. Node 7 is flow dependent on node 4 because: (i) node 4 defines variable `product`, (ii) node 7 references variable `product`, and (iii) there exists a path $4 \rightarrow 5 \rightarrow 6 \rightarrow 7$ without intervening definitions of `product`. Node 7 is control dependent on node 5 because there exists a path $5 \rightarrow 6 \rightarrow 7$ such that: (i) node 6 is post-dominated by node 7, and (ii) node 5 is not post-dominated by node 7.

3 Methods for Static Slicing

3.1 Basic Algorithms for Static Slicing

In this section, we study basic algorithms for static slicing of structured programs without nonscalar variables, procedures, and interprocess communication.

3.1.1 Dataflow Equations

The original definition of program slicing that was introduced by Weiser in [85] is based on iterative solution of dataflow equations[3]. Weiser defines a *slice* as an *executable* program that is obtained from the original program by deleting zero or more statements. A *slicing criterion* consists of a pair (n, V) where n is a node in the CFG of the program, and V a subset of the program's variables. In order to be a slice with respect to criterion (n, V), a subset S of the statements of program P must satisfy the following property: whenever P halts for a given input, S also halts for that input, computing the same values for the variables in V whenever the statement corresponding to node n is executed. At least one slice exists for any criterion: the program itself. A slice is *statement-minimal* if no other slice for the same criterion contains fewer statements. Weiser argues that statement-minimal slices are not necessarily unique, and that the problem of determining statement-minimal slices is undecidable.

Approximations of statement-minimal slices are computed in an iterative process, by computing consecutive sets of *relevant variables* for each node in the CFG. First, the *directly relevant variables* are determined, by only taking data dependences into account. Below, the notation $i \rightarrow_{\text{CFG}} j$ indicates the existence of an edge in the CFG from node i to node j. For a slicing criterion $C \equiv (n, V)$, the set of directly relevant variables at node i of the CFG, $R^0_C(i)$ is defined as follows:

- $R^0_C(i) = V$ when $i = n$.
- For every $i \rightarrow_{\text{CFG}} j$, $R^0_C(i)$ contains all variables v such that either (i) $v \in R^0_C(j)$ and $v \notin \textsc{Def}(i)$, or (ii) $v \in \textsc{Ref}(i)$, and $\textsc{Def}(i) \cap R^0_C(j) \neq \emptyset$.

From this, a set of *directly relevant statements*, S^0_C, is derived. S^0_C is defined as the set of all nodes i which define a variable v that is a relevant at a successor of i in the CFG:

$$S^0_C \equiv \{i \mid \textsc{Def}(i) \cap R^0_C(j) \neq \emptyset, i \rightarrow_{\text{CFG}} j\}$$

Variables referenced in the control predicate of an **if** or **while** statement are *indirectly* relevant, if (at least) one of the statements in its body is relevant. The *range of influence* $\textsc{Infl}(b)$ of a branch statement b is defined as the set of statements that are control dependent on b. The branch statements B^k_C which are relevant due to the influence they have on nodes i in S^k_C are:

$$B^k_C \equiv \{b \mid i \in S^k_C,\ i \in \textsc{Infl}(b)\}$$

[3]Weiser's definition of branch statements with indirect relevance to a slice contains an error [86]. We follow the modified definition proposed in [63]. However, we do not agree with the statement in [63] that "It is not clear how Weiser's algorithm deals with loops".

Node #	Def	Ref	Infl	R_C^0	R_C^1
1	{ n }	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
2	{ i }	$\emptyset$	$\emptyset$	$\emptyset$	{ n }
3	{ sum }	$\emptyset$	$\emptyset$	{ i }	{ i, n }
4	{ product }	$\emptyset$	$\emptyset$	{ i }	{ i, n }
5	$\emptyset$	{ i, n }	{ 6, 7, 8 }	{ product, i }	{ product, i, n }
6	{ sum }	{ sum, i }	$\emptyset$	{ product, i }	{ product, i, n }
7	{ product }	{ product, i }	$\emptyset$	{ product, i }	{ product, i, n }
8	{ i }	{ i }	$\emptyset$	{ product, i }	{ product, i, n }
9	$\emptyset$	{ sum }	$\emptyset$	{ product }	{ product }
10	$\emptyset$	{ product }	$\emptyset$	{ product }	{ product }

Table 1: Results of Weiser's algorithm for the example program of Figure 1 **(a)** and slicing criterion $(10, \texttt{product})$.

The sets of *indirectly relevant variables* R_C^{k+1} are determined by considering the variables in the predicates of the branch statements B_C^k to be relevant.

$$R_C^{k+1}(i) \equiv R_C^k(i) \cup \bigcup_{b \in B_C^k} R^0_{(b, \textsc{Ref}(b))}(i)$$

The sets of *indirectly relevant statements* S_C^{k+1} consist of the nodes in B_C^k together with the nodes i which define a variable that is relevant to a CFG successor j:

$$S_C^{k+1} \equiv B_C^k \cup \{i \mid \textsc{Def}(i) \cap R_C^{k+1}(j) \neq \emptyset, i \rightarrow_{\text{CFG}} j\}$$

The sets R_C^{k+1} and S_C^{k+1} are nondecreasing subsets of the program's variables and statements, respectively; the fixpoint of the computation of the S_C^{k+1} sets constitutes the desired program slice.

As an example, we consider the program of Figure 1 **(a)** and criterion $(10, \texttt{product})$. Table 1 summarizes the Def, Ref, Infl sets, and the sets of relevant variables computed by Weiser's algorithm. The CFG of the program was shown earlier in Figure 3. From the information in the table, and the definition of a slice, we obtain $S_C^0 = \{2, 4, 7, 8\}$, $B_C^0 = \{5\}$, and $S_C^1 = \{1, 2, 4, 5, 7, 8\}$. For our example, the fixpoint of the sets of indirectly relevant variables is reached at set S_C^1. The corresponding slice w.r.t. criterion $C \equiv (10, \texttt{product})$ as computed by Weiser's algorithm is identical to the program shown in Figure 1 **(b)** apart from the fact that the output statement `write(product)` is not contained in the slice.

In fact, an output statement will *never* be part of a slice because: (i) its Def set is empty so that no other statement can either be data dependent on it, and (ii) no statement can be control dependent on an output statement. In [43], Horwitz and Reps suggest a way for making an output value dependent on all previous output values by treating a statement `write(v)` as an assignment `output := output || v`, where `output` is a string-valued variable containing all output of the program, and '`||`' denotes string concatenation. Output statements can be included in the slice by including `output` in the set of variables specified in the criterion.

In [64], Lyle presents a modified version of Weiser's algorithm for computing slices. Apart from some minor changes in terminology, this algorithm is essentially the same as that in [85].

$$
\begin{array}{lcl}
\lambda_{\epsilon} & = & \emptyset \\
\mu_{\epsilon} & = & \emptyset \\
\rho_{\epsilon} & = & \textsc{Id} \\
\\
\lambda_{v:=e} & = & \textsc{Vars}(e) \times \{\, e \,\} \\
\mu_{v:=e} & = & \{\, (e, v) \,\} \\
\rho_{v:=e} & = & (\textsc{Vars}(e) \times \{\, v \,\}) \cup (\textsc{Id} - (v, v)) \\
\\
\lambda_{S_1;S_2} & = & \lambda_{S_1} \cup (\rho_{S_1} \cdot \lambda_{S_2}) \\
\mu_{S_1;S_2} & = & (\mu_{S_1} \cdot \rho_{S_2}) \cup \mu_{S_2} \\
\rho_{S_1;S_2} & = & \rho_{S_1} \cdot \rho_{S_2} \\
\\
\lambda_{\textbf{if } e \textbf{ then } S} & = & (\textsc{Vars}(e) \times \{\, e \,\}) \cup \lambda_S \\
\mu_{\textbf{if } e \textbf{ then } S} & = & (\{\, e \,\} \times \textsc{Defs}(S)) \cup \mu_S \\
\rho_{\textbf{if } e \textbf{ then } S} & = & (\textsc{Vars}(e) \times \textsc{Defs}(S)) \cup \rho_S \cup \textsc{Id} \\
\\
\lambda_{\textbf{if } e \textbf{ then } S_1 \textbf{ else } S_2} & = & (\textsc{Vars}(e) \times \{\, e \,\}) \cup \lambda_{S_1} \cup \lambda_{S_2} \\
\mu_{\textbf{if } e \textbf{ then } S_1 \textbf{ else } S_2} & = & (\{\, e \,\} \times (\textsc{Defs}(S_1) \cup \textsc{Defs}(S_2))) \cup \mu_{S_1} \cup \mu_{S_2} \\
\rho_{\textbf{if } e \textbf{ then } S_1 \textbf{ else } S_2} & = & (\textsc{Vars}(e) \times (\textsc{Defs}(S_1) \cup \textsc{Defs}(S_2))) \cup \rho_{S_1} \cup \rho_{S_2} \cup \textsc{Id} \\
\\
\lambda_{\textbf{while } e \textbf{ do } S} & = & \rho_S^{\star} \cdot ((\textsc{Vars}(e) \times \{\, e \,\}) \cup \lambda_S) \\
\mu_{\textbf{while } e \textbf{ do } S} & = & (\{\, e \,\} \times \textsc{Defs}(S)) \cup \mu_S \cdot \rho_S^{\star} \cdot ((\textsc{Vars}(e) \times \textsc{Defs}(S)) \cup \textsc{Id}) \\
\rho_{\textbf{while } e \textbf{ do } S} & = & \rho_S^{\star} \cdot ((\textsc{Vars}(e) \times \textsc{Defs}(S)) \cup \textsc{Id})
\end{array}
$$

Figure 4: Definition of information-flow relations.

Hausler restates Weiser's algorithm in the style of denotational semantics [36]. In denotational semantics, the behavior of a statement or sequence of statements is characterized by defining how it transforms the state. In *denotational slicing*, a function δ characterizes a language construct by defining how it affects the set of relevant variables (see [85]). Another function, α, uses δ to express how slices can be constructed.

3.1.2 Information-flow Relations

In [16], Bergeretti and Carré define a number of *information-flow relations* for programs which can be used to compute slices. For a statement (or sequence of statements) S, a variable v, and an expression (i.e., a control predicate or the right-hand side of an assignment) e that occurs in S, the relations λ_S, μ_S, and ρ_S are defined. These information-flow relations possess the following properties: $(v, e) \in \lambda_S$ iff the value of v on entry to S potentially affects the value computed for e, $(e, v) \in \mu_S$ iff the value computed for e potentially affects the value of v on exit from S, and $(v, v') \in \rho_S$ iff the value of v on entry to S may affect the value of v' on exit from S. The set E_S^v of *all* expressions e for which $(e, v) \in \mu_S$ can be used to construct *partial statements*. A partial statement of statement S associated with variable v is obtained by replacing all statements in S that do not contain expressions in E_S^v by empty statements.

Information-flow relations are computed in a syntax-directed, bottom-up manner. For an empty statement, the relations λ_S and μ_S are empty, and ρ_S is the identity. For an assignment $v := e$, λ_S contains (v', e) for all variables v' which occur in e, μ_S consists of (e, v), and ρ_S contains (v', v) for all variables which occur in e as well as (v'', v'') for all variables

Expression #[a]	Potentially Affected Variables
1	{ n, sum, product, i }
2	{ sum, product, i }
3	{ sum }
4	{ product }
5	{ sum, product, i }
6	{ sum }
7	{ product }
8	{ sum, product, i }
9	$\emptyset$
10	$\emptyset$

[a] Expression numbers correspond to line numbers in Figure 1 **(a)**.

Figure 5: Information-flow relation μ for the example program of Figure 1 **(a)**.

$v'' \neq v$. Figure 4 shows how information-flow relations for sequences of statements, conditional statements and loop statements are constructed from the information-flow relations of their constituents. In the figure, ϵ denotes an empty statement, "$\cdot$" relational join, ID the identity relation, VARS(e) the set of variables occurring in expression e, and DEFS(S) the set of variables that may be defined in statement S. The convoluted definition for **while** constructs is obtained by effectively transforming it into an infinite sequence of nested one-branch **if** statements. The relation $\rho^\star$ used in this definition is the transitive and reflexive closure of ρ.

A slice w.r.t. the value of a variable v at an arbitrary location can be computed by inserting a dummy assignment $v' := v$ at the appropriate place, where v' is a variable that did not previously occur in S. The slice w.r.t. the final value of v' in the modified program is equivalent to a slice w.r.t. v at the selected location in the original program.

Static *forward* slices can be derived from relation λ_S in a way that is similar to the method for computing static backward slices from the μ_S relation.

Figure 5 shows the information-flow relation μ for the (entire) program of Figure 1 **(a)**[4]. From this relation it follows that the set of expressions which potentially affect the value of `product` at the end of the program are $\{1, 2, 4, 5, 7, 8\}$. The corresponding partial statement is obtained by omitting all statements from the program which do not contain expressions in this set, i.e., both assignments to `sum` and both write statements. The result is exactly the same as the slice computed by Weiser's algorithm (see Section 3.1.1).

3.1.3 Dependence Graph Based Approaches

Ottenstein and Ottenstein were the first of many to define slicing as a reachability problem in a dependence graph representation of a program [69]. They use the Program Dependence Graph (PDG) [27, 58] for static slicing of single-procedure programs. The statements and expressions of a program constitute the vertices of a PDG, and edges correspond to data dependences and control dependences between statements (see Section 2). The key issue is that the partial ordering of the vertices induced by the dependence edges

[4] Bergeretti and Carré do not define information-flow relations for I/O statements. For the purposes of this example, it is assumed that the statement `read(n)` can be treated as an assignment `n := SomeConstant`, and that the statements `write(sum)` and `write(product)` should be treated as empty statements.

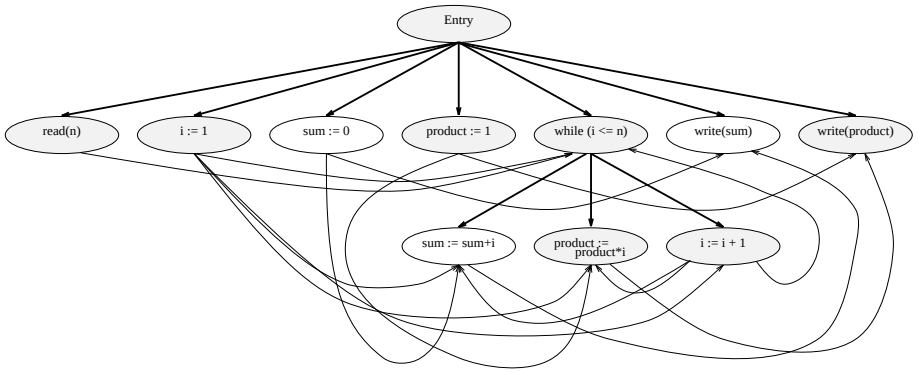


Figure 6: PDG of the program in Figure 1 **(a)**.

must be obeyed so as to preserve the semantics of the program.

In the PDGs of Horwitz et al., a distinction is made between loop-carried and loop-independent flow dependences, and there is an additional type of data dependence edges named *def-order* dependence edges [40, 41, 42, 44]. Horwitz et al. argue that their PDG variant is *adequate*: if two programs have isomorphic PDGS, they are strongly equivalent. This means that, when started with the same input state, they either compute the same values for all variables, or they both diverge. It is argued that the PDG variant of [40] is minimal in the sense that removing any of the dependence edges, or disregarding the distinction between loop-carried and loop-independent flow edges would result in inequivalent programs having isomorphic PDGs. Nevertheless, for the computation of program slices, only flow dependences and control dependences are necessary. We will therefore only consider these dependences in the sequel.

In all dependence graph based approaches, the slicing criterion is identified with a vertex v in the PDG. In Weiser's terminology, this corresponds to a criterion (n, V) where n is the CFG node corresponding to v, and V the set of *all* variables defined or used at v. Consequently, slicing criteria of PDG-based slicing methods are less general than those of methods based on dataflow equations or information-flow relations. However, in Section 3.6.2, we will discuss how more precise slicing criteria can be 'simulated' by PDG-based slicing methods. For single-procedure programs, the slice w.r.t. v consists of all vertices from which v is reachable. The related parts of the source text of the program can be found by maintaining a mapping between vertices of the PDG and the source text during the construction of the PDG.

The PDG variant of [69] shows considerably more detail than that of [44]. In particular, there is a vertex for each (sub)expression in the program, and file descriptors appear explicitly as well. As a result, `read` statements involving irrelevant variables are not 'sliced away', and slices will execute correctly with the full input of the original program.

As an example, Figure 6 shows the PDG of the program of Figure 1 **(a)**. In this figure, the PDG variant of [44] is used. Thick edges represent control dependences[5] and

[5]We omit the usual labeling of control dependence edges, as this is irrelevant for the present discussion. Furthermore, we will omit loop-carried flow dependence edges from a vertex to itself, as such edges are irrelevant for the computation of slices.

```
program Example;
begin
  a := 17;
  b := 18;
  P(a, b, c, d);
  write(d)
end

procedure P(v, w, x, y);
  x := v;
  y := w
end
```
(a)

```
program Example;
begin
  a := 17;
  b := 18;
  P(a, b, c, d);

end

procedure P(v, w, x, y);
        ;
  y := w
end
```
(b)

```
program Example;
begin
        ;
  b := 18;
  P(a, b, c, d);
  write(d)
end

procedure P(v, w, x, y);
        ;
  y := w
end
```
(c)

Figure 7: **(a)** Example program. **(b)** Weiser's slice. **(a)** HRB slice.

thin edges represent flow dependences. Shading is used to indicate the vertices in the slice w.r.t. `write(product)`.

3.2 Interprocedural Static Slicing

3.2.1 Dataflow Equations

Weiser describes a two-step approach for computing interprocedural static slices in [85, 86]. First, a slice is computed for the procedure P which contains the original slicing criterion. The effect of a procedure call on the set of relevant variables is approximated using interprocedural summary information [13]. For a procedure P, this information consists of a set $\textsc{Mod}(P)$ of variables that may be modified by P, and a set $\textsc{Use}(P)$ of variables that may be used by P, taking into account any procedures called by P. A call to P is treated as though it defines all variables in $\textsc{Mod}(P)$ and uses all variables in $\textsc{Use}(P)$, where actual parameters are substituted for formal parameters [86]. The fact that Weiser's algorithm does not take into account *which* output parameters are dependent on *which* input parameters is a cause of imprecision. Figure 7 **(a)** shows an example program that manifests this problem. The interprocedural slicing algorithm of [85] will compute the slice shown in Figure 7 **(b)**. This slice contains the statement `a := 17` due to the spurious dependence between variable `a` before the call, and variable `d` after the call. The Horwitz-Reps-Binkley algorithm that will be discussed in Section 3.2.3 will compute the more accurate slice shown in Figure 7 **(c)**.

In the second step of Weiser's algorithm for interprocedural slicing, new criteria are generated for (i) procedures Q called by P, and (ii) procedures R that call P. The two steps are repeated until no new criteria occur. The new criteria of (i) consist of all pairs (n_Q, V_Q) where n_Q is the last statement of Q and V_Q is the set of relevant variables in P which is in the scope of Q (formals are substituted for actuals). The new criteria of (ii) consist of all pairs (N_R, V_R) such that N_R is a call to P in R, and V_R is the set of relevant variables at the first statement of P which is in the scope of R (actuals are substituted for formals). The generation of new criteria is formalized by way of functions $\textsc{Up}(\mathcal{S})$ and $\textsc{Down}(\mathcal{S})$ which map a set $\mathcal{S}$ of slicing criteria in a procedure P to a set of criteria in procedures that call P, and a set of criteria in procedures called by P, respectively. The closure $(\textsc{Up} \cup \textsc{Down})*(\{C\})$ contains all criteria necessary to compute an interprocedural slice, given an initial criterion C. Worst-case assumptions have to be made when a program

```
     program Example;
     begin
(1)  read(n);
(2)  i := 1;
(3)  sum := 0;
(4)  product := 1;
(5)  while i <= n do
     begin
(6)    Add(sum, i);
(7)    Multiply(product, i);
(8)    Add(i, 1)
     end;
(9)  write(sum);
(10) write(product)
     end

     procedure Add(a; b);
     begin
(11) a := a + b
     end

     procedure Multiply(c; d);
     begin
(12) j := 1;
(13) k := 0;
(14) while j <= d do
     begin
(15)   Add(k, c);
(16)   Add(j, 1);
     end;
(17) c := k
     end
```

Figure 8: Example of a multi-procedure program.

calls external procedures, and the source-code is unavailable.

For example, assume that a slice is to be computed w.r.t. the final value of `product` in the program of Figure 8. Slicing will begin with the initial criterion $(10, \texttt{product})$. The first step of Weiser's algorithm will include all lines of the main program except line 3 and 6. In particular, the procedure calls `Multiply(product, i)` and `Add(i, 1)` are included in the slice, because: (i) the variables `product` and `i` are deemed relevant at those points, and (ii) using interprocedural data flow analysis it can be determined that $\textsc{Mod}(\texttt{Add}) = \{\, a \,\}$, $\textsc{Use}(\texttt{Add}) = \{\, a,\ b \,\}$, $\textsc{Mod}(\texttt{Multiply}) = \{\, c \,\}$, and $\textsc{Use}(\texttt{Multiply}) = \{\, c,\ d \,\}$. As the initial criterion is in the main program, we have $\textsc{Up}(\{\, (10, \texttt{product}) \,\}) = \emptyset$; $\textsc{Down}(\{\, (10, \texttt{product}) \,\})$ contains the criteria $(11, \{\, \texttt{a} \,\})$ and $(17, \{\, \texttt{c}, \texttt{d} \,\})$. The result of slicing procedure `Add` with criterion $(11, \{\, \texttt{a} \,\})$ and procedure `Multiply` with criterion $(17, \{\, \texttt{c}, \texttt{d} \,\})$ will be the inclusion of these procedures in their entirety. Note that the calls to `Add` at lines 15 and 16 causes the generation of a new criterion $(11, \{\, \texttt{a}, \texttt{b} \,\})$ and thus re-slicing of procedure `Add`.

Horwitz, Reps, and Binkley report in [44] that Weiser's algorithm for interprocedural slicing is unnecessarily inaccurate, because of what they refer to as the 'calling context' problem. In a nutshell, the problem is that when the computation 'descends' into a procedure Q that is called from a procedure P, it will 'ascend' to *all* procedures that call Q, not only P. This corresponds to execution paths which enter Q from P and exit Q to a different procedure P'. These execution paths are infeasible; taking them into consideration results in inaccurate slices. The example of Figure 8 exhibits the 'calling context' problem. Since line (11) is in the slice, new criteria are generated for *all* calls to `Add`. These calls include the (already included) calls at lines 8, 15, and 16, but also the call `Add(sum, i)` at line 6. The new criterion $(6, \{\, \texttt{sum}, \texttt{i} \,\})$ that is generated will cause the inclusion of lines 6 and 3 in the slice. Consequently, the slice consists of the entire program.

We conjecture that the calling context problem of Weiser's algorithm can be fixed by observing that the criteria in the Up sets are only needed to include procedures that

```
program Main;
...
  while ( ··· ) do
    P(x1, x2, ···, xn);
    z := x1;
    x2 := x1;
    x3 := x2;
    ...
    xn := x(n-1)
  end;
  write(z)
end

procedure P( y1, y2, ···, yn );
begin
  write(y1);
  write(y2);
  ...
  write(yn)
end
```

Figure 9: Example program where procedure `P` is sliced n times by Weiser's algorithm.

(transitively) call the procedure containing the initial criterion[6]. Once this is done, *only* DOWN sets need to be computed. Reps suggested that this essentially corresponds to the two passes of the Horwitz-Reps-Binkley algorithm (see Section 3.2.3) if all UP sets are computed before determining any DOWN sets.

The computation of the UP and DOWN sets requires that the sets of relevant variables are known at all call sites. In other words, the computation of these sets involves *slicing* of procedures. In the course of doing this, new variables may become relevant at previously encountered call sites, and new call sites may be encountered. This is illustrated by the program shown in Figure 9. In the subsequent discussion, L denotes the line-number of statement `write(z)` and M the line-number of the last statement in procedure `P`. Computing the slice w.r.t. criterion $(L, \{$ `z` $\})$ requires n iterations of the body of the **while** loop. During the i^{th} iteration, variables $x_1, \cdots, x_i$ will be relevant at the call site, causing the inclusion of criterion $(M, \{ y_1, \cdots, y_i \})$ in DOWN(`Main`). If no precaution is taken to combine the criteria in DOWN(`Main`), procedure `P` will be sliced n times.

Hwang, Du, and Chou propose an iterative solution for interprocedural static slicing based on replacing recursive calls by instances of the procedure body in [45]. The slice is recomputed in each iteration until a fixed point is found, i.e., no new statements are added to a slice. This approach does not suffer from the calling context problem because expansion of recursive calls does not lead to considering infeasible execution paths. However, Reps has shown recently that for a certain family P^k of recursive programs, this algorithm takes time $O(2^k)$, i.e., exponential in the length of the program [72, 74]. An example of such a program is shown in Figure 10 **(a)**. Figure 10 **(b)** shows the exponentially long path that is effectively traversed by the Hwang-Du-Chou algorithm.

3.2.2 Information-flow Relations

In [16], Bergeretti and Carré explain how the effect of procedure calls can be approximated. Exact dependences between input and output parameters are determined by slicing the called procedure with respect to each output parameter (i.e., computation of the μ relation for the procedure). Then, each procedure call is replaced by a set of assignments, where each output parameter is assigned a fictitious expression that contains the input parame-

[6] A similar observation was made by Jiang et al. in [47]. However, they do not explain that this approach only works when a call to procedure p is treated as a multiple assignment MOD(p) := USE(p).

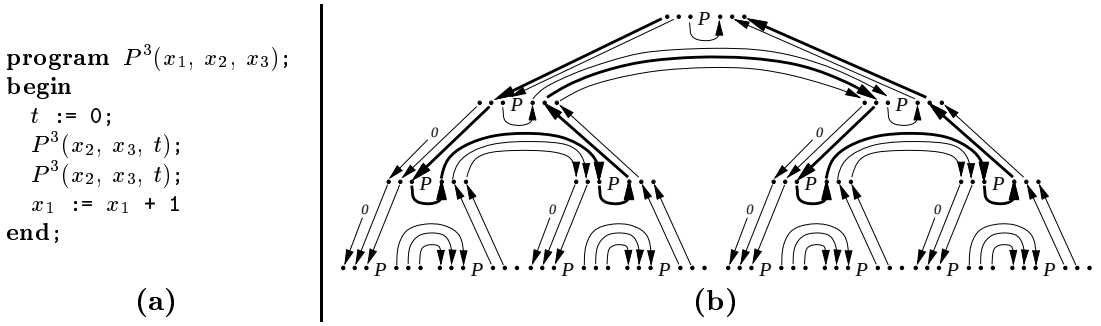


Figure 10: **(a)** Example program. **(b)** Exponentially long path traversed by the Hwang-Du-Chou algorithm for interprocedural static slicing.

ters it depends upon. As only feasible execution paths are considered, this approach does not suffer from the calling context problem. A call to a side-effect free function can be modeled by replacing it with a fictitious expression containing all actual parameters. Note that the computed slices are not truly interprocedural since no attempt is done to slice procedures other than the main program.

For the example program of Figure 8, the slice w.r.t. the final value of `product` would include all statements except `sum := 0`, `Add(sum,i)`, and `write(sum)`.

3.2.3 Dependence Graphs

Horwitz, Reps, and Binkley introduce the notion of a *System Dependence Graph* (SDG) for slicing of multi-procedure programs [44]. Parameter passing by value-result is modeled as follows: (i) the calling procedure copies its actual parameters to *temporary* variables before the call, (ii) the formal parameters of the called procedure are initialized using the corresponding temporary variables, (iii) before returning, the called procedure copies the final values of the formal parameters to the temporary variables, and (iv) after returning, the calling procedure updates the actual parameters by copying the values of the corresponding temporary variables.

An SDG contains a program dependence graph for the main program, and a procedure dependence graph for each procedure. There are several types of vertices and edges in SDGs which do not occur in PDGs. For each call statement, there is a *call-site vertex* in the SDG as well as *actual-in* and *actual-out* vertices which model the copying of actual parameters to/from temporary variables. Each procedure dependence graph has an entry vertex, and *formal-in* and *formal-out* vertices to model copying of formal parameters to/from temporary variables. Actual-in and actual-out vertices are control dependent on the call-site vertex; formal-in and formal-out vertices are control dependent on the procedure's entry vertex. In addition to these *intra*procedural dependence edges, an SDG contains the following *inter*procedural dependence edges: (i) a control dependence edge between a call-site vertex and the entry vertex of the corresponding procedure dependence graph, (ii) a *parameter-in* edge between corresponding actual-in and formal-in vertices, (iii) a *parameter-out* edge between corresponding formal-out and actual-out vertices, and (iv) edges which represent *transitive inter*procedural data dependences. These transitive dependences are computed by constructing an attribute grammar based on the call graph of the system, and serve to circumvent the calling context problem. This is accomplished

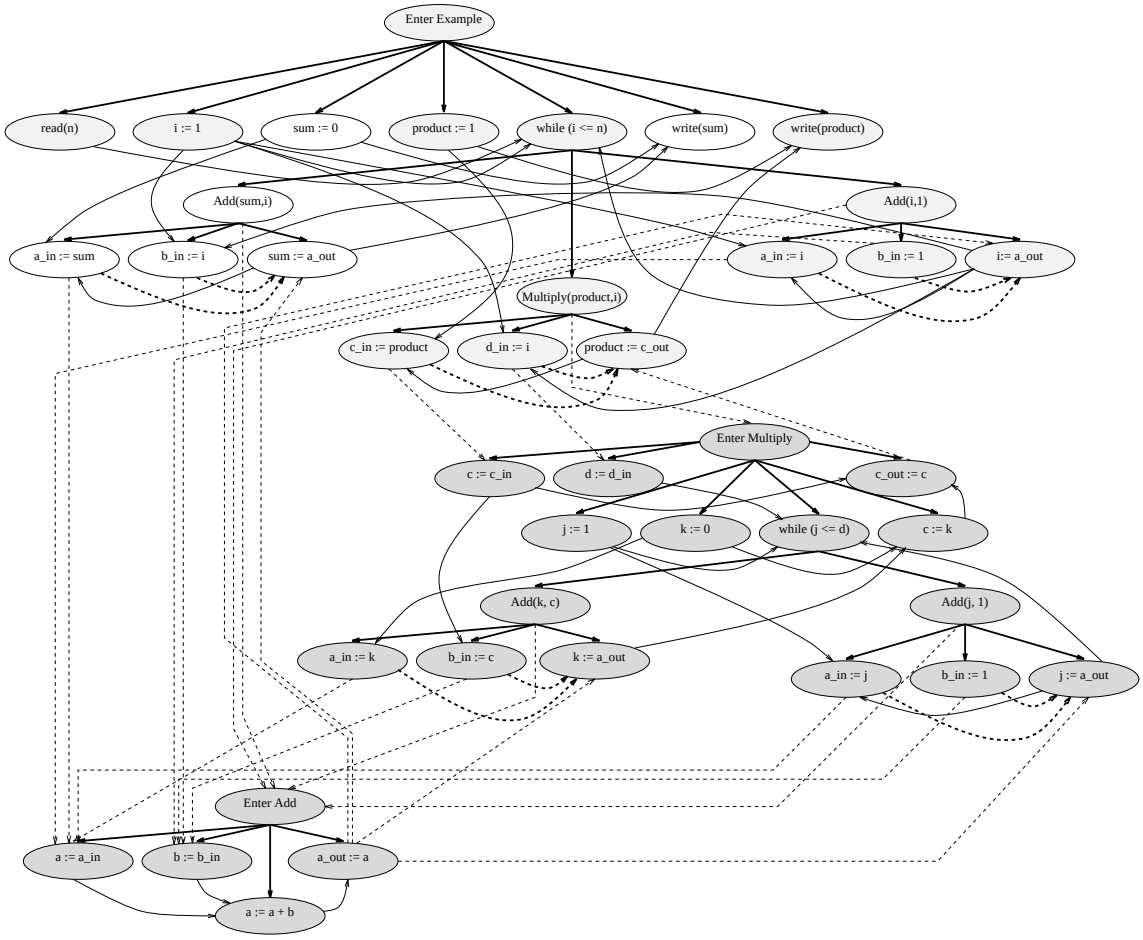


Figure 11: SDG of the program in Figure 8.

by traversing the graph in two phases. Suppose that slicing starts at vertex s. The first phase determines all vertices from which s can be reached *without descending into procedure calls*. The transitive interprocedural dependence edges guarantee that calls can be side-stepped, without descending into them. In the second phase, the algorithm descends into all previously side-stepped calls and determines the remaining vertices in the slice.

Using interprocedural data flow analysis [11], the sets of variables which can be referenced or modified by a procedure can be determined. This information can be used to eliminate actual-out and formal-out vertices for parameters that will never be modified, resulting in more precise slices. The algorithm of [44] also works for call-by-reference parameter passing[7] provided that aliases are resolved. Two approaches are proposed: transformation of the original program into an equivalent alias-free program, or the use of a generalized flow dependence notion (as will be discussed in Section 3.4). The first approach yields more precise slices, whereas the second is more efficient.

Figure 11 shows the SDG for the program of Figure 8 where interprocedural dataflow analysis is used to eliminate the vertices for the second parameters of the procedures `Add` and `Multiply`. In the figure, thin solid arrows represent flow dependences, thick solid ar-

[7] For a discussion of parameter passing mechanisms the reader is referred to [6], Section 7.5.

rows correspond to control dependences, thin dashed arrows are used for call, parameter-in, and parameter-out dependences, and thick dashed arrows represent transitive interprocedural flow dependences. The vertices in the slice w.r.t. statement `write(product)` are shown shaded; light shading indicates the vertices identified in the first phase of the algorithm, and dark shading indicates the vertices identified in the second phase. Clearly, the statements `sum := 0`, `Add(sum, i)`, and `write(sum)` are not in the slice.

Slices computed by the algorithm of [44] are not necessarily executable programs. Cases where only a subset of the vertices for actual and formal parameters are in the slice, correspond to procedures where *some* of the arguments are 'sliced away'; for different calls to the procedure, different arguments may be omitted. Horwitz et al. propose two methods for transforming such a non-executable slice into an executable program. The first method consists of creating different variants of a procedure in the slice, and has the disadvantage that the slice is no longer a restriction of the original program. The second solution consists of extending the slice with all parameters that are present at *some* call to *all* calls which occur in the slice. In addition, all vertices on which the added vertices are dependent must be added to the slice as well. Clearly, this second approach has the disadvantage of yielding larger slices.

Finally, it is outlined how interprocedural slices can be computed from partial SDGs (corresponding to programs under development, or programs containing library calls) and how, using the SDG, interprocedural *forward* slices can be computed in a way that is very similar to the previously described method for interprocedural (backward) slicing.

Recently, Reps et al. proposed a new algorithm for computing the summary edges of an SDG [74, 75], which is asymptotically more efficient than the Horwitz-Reps-Binkley algorithm [44] (the time requirements of these algorithms will be discussed in Section 3.6.3). Input to the algorithm is an SDG where no summary edges have been added yet, i.e., a collection of procedure dependence graphs connected by call, parameter-in, and parameter-out edges. The algorithm uses a worklist to determine *same-level* realizable paths. Intuitively, a same-level realizable path obeys the call-return structure of procedure calls, and it starts and ends at the same level (i.e., in the same procedure). Same-level realizable paths between formal-in and formal-out vertices of a procedure P induce summary edges between the corresponding actual-in and actual-out vertices for any call to P. The algorithm starts by asserting that a same-level realizable path of length zero exists from any formal-out vertex to itself. A worklist is used to select a path, and extend it by adding an edge to its beginning. In [75], a demand-version of the algorithm is presented, which *incrementally* determines the summary edges of an SDG.

In [60], Lakhotia presents an algorithm for computing interprocedural slices that is also based on SDGs. This algorithm computes slices that are identical to the slices computed by the algorithm in [44]. Associated with every SDG vertex v is a three-valued tag; possible values for this tag are: "$\bot$" indicating that v has not been visited, "$\top$" indicating that v has been visited, and *all* vertices from which v can be reached should be visited, and "β" indicating that v has been visited, and *some* of the vertices from which v can be reached should be visited. More precisely, an edge from an entry vertex to a call vertex should only be traversed if the call vertex is labeled $\top$. A worklist algorithm is used to visit all vertices labeled $\top$ before visiting any vertex labeled β. When this process ends, vertices labeled either $\top$ or β are in the slice. Lakhotia's algorithm traverses performs a single pass through the SDG. However, unlike the algorithm of [44], the value of a tag may change *twice*. Therefore it is unclear if Lakhotia's algorithm is really an improvement over the

Horwitz-Reps-Binkley two-pass traversal algorithm.

3.3 Static Slicing in the Presence of Unstructured Control Flow

3.3.1 Dataflow Equations

Lyle reports in [64] that (his version of) Weiser's algorithm for static slicing yields incorrect slices in the presence of unstructured control flow: the behavior of the slice is not necessarily a projection of the behavior of the program. He presents a conservative solution for dealing with **goto** statements consisting of including any **goto** which has a non-empty set of active variables associated with it.

Gallagher [31] and Gallagher and Lyle [32] also use a variation of Weiser's method. A **goto** statement is included in the slice if it jumps to a label of an included statement[8]. Agrawal shows in [2] that this algorithm does not produce correct slices in all cases.

Jiang et al. extend Weiser's slicing method to C programs with arbitrary control flow [47]. They introduce a number of additional rules to 'collect' the unstructured control flow statements such as **goto**, **break**, and **continue** which are part of the slice. Unfortunately, no formal justification is given for the treatment of unstructured control flow constructs in [47]. Agrawal shows in [2] that this algorithm may also produce incorrect slices.

3.3.2 Dependence Graphs

Ball and Horwitz [8, 9] and Choi and Ferrante [21] discovered independently that conventional PDG-based slicing algorithms produce incorrect results in the presence of unstructured control flow: slices may compute values at the criterion that differ from what the original program does. These problems are due to the fact that the algorithms do not determine correctly when unconditional jumps such as **break**, **goto**, and **continue** statements are required in a slice.

As an example, Figure 12 **(a)** shows a variant of our example program which uses a **goto** statement. Figure 12 **(b)** shows the PDG for this program. The vertices which have a transitive dependence on statement `write(product)` are highlighted. Figure 12 **(c)** shows a textual representation of the program thus obtained. Clearly, this 'slice' is incorrect because it does not contain the **goto** statement, causing non-termination. In fact, the previously described PDG-based algorithms will *only* include a **goto** if it is the slicing criterion itself, because no statement is either data or control dependent on a **goto**.

The solution of [8, 9] and the first solution presented in [21] are remarkably similar: unconditional jumps are regarded as *pseudo-predicate* vertices where the 'true' branch consists of the statement that is being jumped to, and the 'false' branch of the *textually* next statement. Correspondingly, there are two outgoing edges in the *augmented* control flow graph (ACFG). Only one of these edges can actually be traversed during execution; the other outgoing edge is 'non-executable'. The notion of (data) flow dependence is altered in order to ignore dependences caused by non-executable edges. Augmented PDGs are constructed using the ACFG instead of the CFG, and slicing is defined in the usual way

[8] Actually, this is a slight simplification. Each basic block is partitioned into labeled blocks; a *labeled block* is a subsequence of the statements in a basic block starting with a labeled statement, and containing no other labeled statements. A **goto** is included in the slice if it jumps to a label for which there is some included statement in its block.

```
   read(n);
   i := 1;
   sum := 0;
   product := 1;
   while true do
   begin
       if (i > n) then
           goto L;
       sum := sum + i;
       product := product * i;
       i := i + 1
   end;
L: write(sum);
   write(product)
```

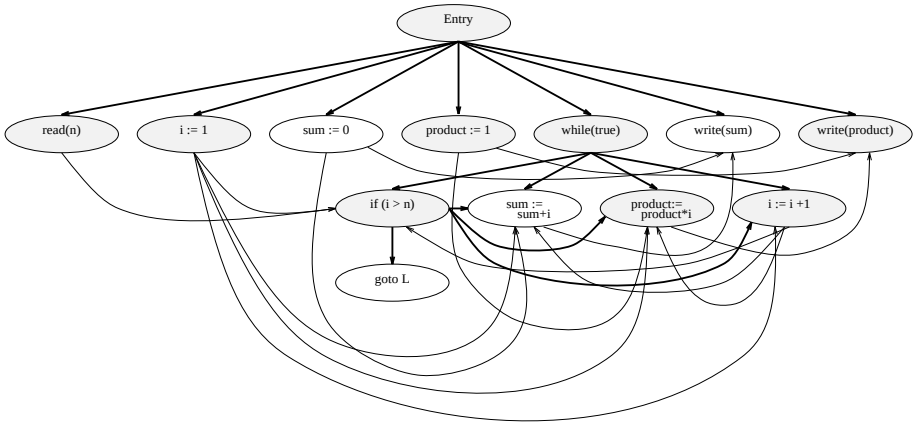
(a)

```
   read(n);
   i := 1;

   product := 1;
   while true do
   begin
       if (i > n) then
                 ;

       product := product * i;
       i := i + 1
   end;

   write(product)
```

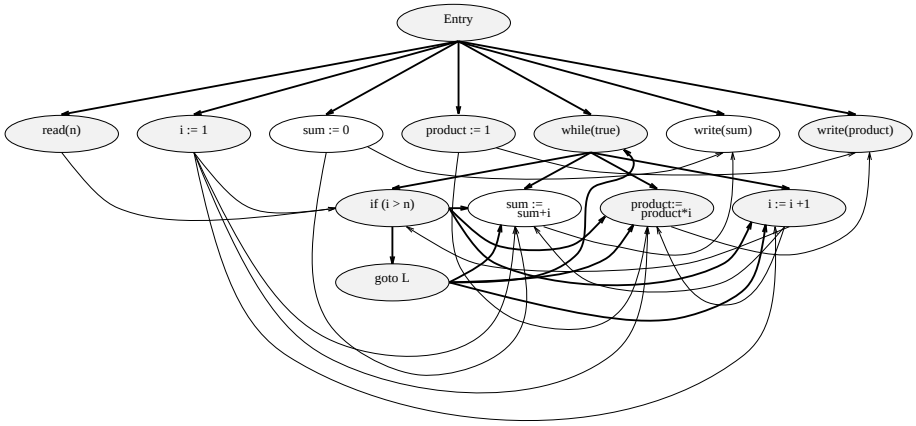
(c)

```
   read(n);
   i := 1;

   product := 1;
   while true do
   begin
       if (i > n) then
           goto L;

       product := product * i;
       i := i + 1
   end;
L:
   write(product)
```

(e)



(b)



(d)

Figure 12: **(a)** Program with unstructured control flow, **(b)** PDG for program of **(a)**, **(c)** incorrect slice, **(d)** Augmented PDG for program of **(a)**, **(e)** correct slice.

as a graph reachability problem. Labels pertaining to statements excluded from the slice are moved to the closest post-dominating statement that occurs in the slice.

The main difference between the approach by Ball and Horwitz and the first approach of Choi and Ferrante is that the latter use a slightly more limited example language: conditional and unconditional **goto**'s are present, but no structured control flow constructs. Although Choi and Ferrante argue that these constructs can be transformed into conditional and unconditional **goto**'s, Ball and Horwitz show that, for certain cases, this results in overly large slices. Both groups present a formal proof that their algorithms compute correct slices.

Figure 12 **(d)** shows the augmented PDG for the program of Figure 12 **(a)**; vertices from which the vertex labeled `write(product)` can be reached are indicated by shading. The (correct) slice corresponding to these vertices is shown in Figure 12 **(e)**.

Choi and Ferrante distinguish two disadvantages of the slicing approach based on augmented PDGs. First, APDGs require more space than conventional PDGs, and their construction takes more time. Second, non-executable control dependence edges give rise to spurious dependences in some cases. In their second approach, Choi and Ferrante utilize the 'classical' PDG. As a first approximation, the standard algorithm for computing slices is used, which by itself produces incorrect results in the presence of unstructured control flow. The basic idea is that for each statement that is *not* in the slice, a new **goto** to its immediate post-dominator is added. In a separate phase, redundant cascaded **goto** statements are removed. The second approach has the advantage of computing smaller slices than the first. A disadvantage, however, is that slices may include **goto** statements which do not occur in the original program.

Yet another PDG-based method for slicing programs with unstructured control flow was recently proposed by Agrawal in [2]. Unlike the methods in [8, 9, 21], Agrawal uses the standard PDG. He observes that a *conditional* jump statement of the form **if** `P` **then goto** `L` must be included in the slice if predicate `P` is in the slice because another statement in the slice is control dependent on it. The terminology 'conventional slicing algorithm' is adopted to refer to the standard PDG-based slicing method, with the above extension to conditional jump statements.

The main observation in [2] is that an unconditional jump statement J should be included in the slice if and only if the immediate postdominator of J *in the slice* differs from the immediate lexical successor of J *in the slice*. Here, a statement S' is a *lexical successor* of a statement S if S textually precedes S' in the program[9]. The statements on which the newly added statement is transitively dependent must also be added to the slice. The motivation for this approach can be understood by considering a sequence of statements $S_1; S_2; S_3$ where S_1 and S_3 are in the slice, and where S_2 contains an unconditional jump statement to a statement that does not have S_3 as its lexical successor. Suppose that S_2 were not included in the slice. Then the flow of control in the slice would pass unconditionally from S_1 to S_3, though in the original program this need not always be the case, because the jump might transfer the control elsewhere. Therefore S_2 must be included, together with all statements it depends upon. Agrawal's algorithm traverses the postdominator tree of a program in pre-order, and considers jump statements for inclusion in this order. The algorithm iterates until no jump statements can be added; this

[9] As Agrawal observes, this notion is equivalent to the non-executable edges in the augmented control flow graphs used in [8, 9, 21].

is necessary because adding a jump (and the statements it depend upon) may change the lexical successors and postdominators *in the slice* of other jump statements, which may therefore need to be included as well. Although no proof is stated, Agrawal claims that his algorithm computes correct slices, and that it computes slices that are identical to those computed by the algorithm in [8, 9].

The algorithm in [2] may be simplified significantly if the only type of jump that occurs in a program is a *structured jump*, i.e., a jump to a lexical successor. C **break**, **continue**, and **return** statements are all structured jumps. First, only a single traversal of the post-dominator tree is required. Second, jump statements have to be added only if they are control dependent on a predicate that is in the slice. In this case, the statements they are dependent upon are already included in the slice. For programs with structured jumps, the algorithm can be further simplified to a *conservative* algorithm by including in the slice all jump statements that are control dependent on a predicate that is in the slice.

Agrawal's algorithm will include the **goto** statement of the example program of Figure 12 **(a)** because it is control dependent on the (included) predicate of the **if** statement.

3.4 Static Slicing in the Presence of Composite Datatypes/Pointers

Lyle proposes a conservative solution to the problem of static slicing in the presence of arrays [64]. Essentially, any update to an element of an array is regarded as an update and a reference of the entire array.

The PDG variant of Ottenstein and Ottenstein [69] contains a vertex for each sub-expression; special *select* and *update* operators serve to access elements of an array.

In the presence of pointers (and procedures), situations may occur where two or more variables refer to the same memory location; this phenomenon is commonly called *aliasing*. Algorithms for determining potential aliases can be found in [20, 61]. Slicing in the presence of aliasing requires a generalization of the notion of data dependence to take potential aliases into account. This can be done roughly as follows: a statement s is *potentially data dependent* on a statement s' if: (i) s defines a variable X', (ii) s' uses a variable X, (iii) X and X' are potential aliases, and (iv) there exists a path from s to s' in the CFG where X is not necessarily defined. Such paths may contain definitions to potential aliases of X. This altered notion of data dependence can in principle be used in any static slicing algorithm.

A slightly different approach is pursued by Horwitz, Pfeiffer, and Reps in [38]. Instead of defining data dependence in terms of potential definitions and uses of variables, they defined this notion in terms of potential definitions and uses of *abstract memory locations*. The PDG-based static slicing algorithm proposed by Agrawal, DeMillo and Spafford [3] implements a similar idea to deal with both composite variables and pointers.

Reaching definitions for a scalar variable v at node n in the flowgraph are determined by finding all paths from nodes corresponding to a definition of v to n which do not contain other definitions of v. When composite datatypes and pointers are considered, definitions involve *l-valued expressions* rather than variables. An l-valued expression is any expression which may occur as the left-hand side of an assignment. For composite datatypes and pointers, a new definition of reaching definitions is presented which is based on the layout of memory locations occupied by l-valued expressions rather than on names of variables. Memory locations are regarded as abstract quantities (e.g., the array a corresponds to 'locations' a[1], a[2],$\cdots$). Whereas a definition for a scalar variable either does or does not

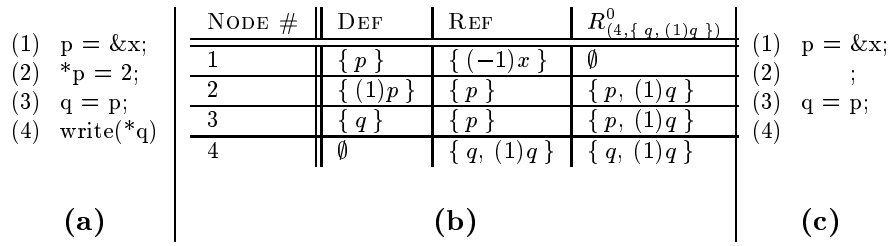


Figure 13: **(a)** Example program. **(b)** Defined variables, used variables, and relevant variables for this program. **(c)** Incorrect slice.

reach a use, the situation becomes more complex when composite datatypes and pointers are allowed. For a def-expression e_1 and a use-expression e_2, the following situations may occur:

- *Complete Intersection*
  The memory locations corresponding to e_1 are a superset of the memory locations corresponding to e_2. An example is the case where e_1 defines the whole of record b, and e_2 is a use of $b.f$.

- *Maybe Intersection*
  It cannot be determined statically whether or not the memory locations of a e_1 coincide with those of e_2. This situation occurs when e_1 is an assignment to array element $a[i]$ and e_2 is a use of array element $a[j]$. Pointer dereferencing may also give rise to Maybe intersections.

- *Partial Intersection*
  The memory locations of e_1 are a subset of the memory locations of e_2. This occurs for example when array a record is used at e_2, and array element $a[5]$ is defined at e_1.

Given these concepts, an extended reaching definition function is defined which traverses the flowgraph until it finds Complete Intersections, makes worst-case assumptions in the case of Maybe Intersections, and continues the search for the array or record parts which have not been defined yet in the case of Partial Intersections.

Jiang, Zhou and Robson present an algorithm in [47] for slicing C programs with pointers and arrays. Weihl's notion of *dummy variables* is used for addresses that may be pointed to [81]; Unfortunately, the approach by Jiang et al. appears to be flawed. Figure 13 **(a)** shows an example program, Figure 13 **(b)** the Def, Ref, and R^0_C sets for each statement, and Figure 13 **(c)** the incorrect slice computed by the algorithm of [47] for criterion $C = (4, \{ q, (1)q \})$. In Figure 13 **(b)**, the dummy variables $(1)p$ and $(1)q$ denote the values pointed to by p and q, respectively, and $(-1)x$ denotes the address of x. The second statement is incorrectly omitted because it does not define any variable that is relevant at statement 3.

3.5 Static Slicing of Distributed Programs

In [19], Cheng considers static slicing of concurrent programs using dependence graphs. He generalizes the notions of a CFG and a PDG to a *nondeterministic parallel control flow*

net, and a *program dependence net* (PDN), respectively. In addition to edges for data dependence and control dependence, PDNs may also contain edges for selection dependences, synchronization dependences, and communication dependences. *Selection* dependence is similar to control dependence but involves nondeterministic selection statements, such as the ALT statement of Occam-2. *Synchronization* dependence reflects the fact that the start or termination of the execution of a statement depends on the start or termination of the execution of another statement. *Communication* dependence corresponds to situations where a value computed at one point in the program influences the value computed at another point through interprocess communication. Static slices are computed by solving a reachability problem in a PDN. Unfortunately, Cheng does not clearly state the semantics of synchronization and communication dependence, nor does he state or prove any property of the slices computed by his algorithm.

An interesting point is that Cheng uses a notion of *weak* control dependence [70] for the construction of PDNs. This notion subsumes the standard notion of control dependence; the difference is that weak control dependences exist between the control predicate of a loop, and the statements that follows it. For example, the statements on lines 9 and 10 of the program of Figure 1 **(a)** are weakly control dependent (but not strongly control dependent) on the control predicate of the **while** statement on line 5.

3.6 Comparison of Methods for Static Slicing

3.6.1 Overview

In this section, we compare and classify the static slicing methods that were presented earlier. The section is organized as follows: Section 3.6.1 summarizes the problems that are addressed in the literature. Sections 3.6.2 and 3.6.3 compare the *accuracy* and *efficiency* of slicing methods that address the same problem, respectively. Finally, in Section 3.6.4 we discuss the possibilities for combining algorithms that deal with different problems.

Table 2 provides an overview of the most significant slicing algorithms that can be found in the literature. For each paper, the table lists the computation method used and indicates: (i) whether or not interprocedural slices can be computed, (ii) the control flow constructs under consideration, (iii) the datatypes under consideration, and (iv) whether or not interprocess communication is considered. It is important to realize that the entries of Table 2 only indicate the *problems that have been addressed*; the table does *not* indicate the 'quality' of the solutions (with the exception that incorrect solutions are indicated by footnotes). Moreover, the table also does *not* indicate which algorithms may be combined. For example, the interprocedural slicing algorithm of [44] could in principle be combined with any of the dependence graph based slicing methods for dealing with arbitrary control flow [2, 9, 21]. Possibilities for such combinations are investigated in Section 3.6.4.

In [48], Kamkar distinguishes between methods for computing slices that are executable programs, and those for computing slices that consist of a set of 'relevant' statements and control predicates. We agree with the observation by Horwitz et al. in [44], that for *static* slicing of *single-procedure* programs this is merely a matter of presentation. As we remarked in Section 3.2.3, for static slicing of *multi-procedure* programs, the distinction between executable and non-executable slices is relevant. However, since these problems are strongly related (the solution to the former problem can be used to obtain a solution to the latter problem), we believe the distinction between executable and non-executable

	Computation Method[a]	Interprocedural Solution	Control Flow[b]	Data Types[c]	Interprocess Communication
Weiser [63, 85]	D	yes	S	S	no
Lyle [64]	D	no	A	S, A	no
Gallagher, Lyle [31, 32]	D	no	A[d]	S	no
Jiang et al. [47]	D	yes	A[d]	S, A, P[e]	no
Hausler [36]	F	no	S	S	no
Bergeretti, Carré [16]	I	yes[f]	S	S	no
Ottenstein [69]	G	no	S	S, A	no
Horwitz et al. [41, 42, 76]	G	no	S	S	no
Horwitz et al. [44]	G	yes	S	S	no
Reps et al. [75]	G	yes	S	S	no
Lakhotia [60]	G	yes	S	S	no
Agrawal et al. [3]	G	no	S	S, A, P	no
Ball, Horwitz [8, 9]	G	no	A	S	no
Choi, Ferrante [21]	G	no	A	S	no
Agrawal [2]	G	no	A	S	no
Cheng [19]	G	no	S	S	yes

[a] D = dataflow equations, F = functional/denotational semantics, I = information-flow relations, G = reachability in a dependence graph.

[b] S = structured, A = arbitrary.

[c] S = scalar variables, A = arrays/records, P = pointers.

[d] Solution incorrect (see [2]).

[e] Solution incorrect (see Section 3.4).

[f] Non-recursive procedures only.

Table 2: Overview of static slicing methods.

static slices can be dismissed.

3.6.2 Accuracy

The following issues complicate the comparison of the static slicing methods:

- In its original formulation, Weiser's slicing algorithm [85] considers each line of source code as a unit; this may result in imprecise slices if a line contains more than one statement. Algorithms based on information-flow relations [16] and PDGs [69] do not suffer from this problem because each statement is a distinct unit.

  In subsequent discussions, we will feel free to ignore this fact because one can easily imagine a reformulation of Weiser's algorithm that is based on labeled expressions (as in [16]) instead of line-numbers.

- For slicing methods based on dataflow equations and information-flow relations, a slicing criterion consists of a pair (s, V), where s is a statement and V an arbitrary set of variables. In contrast, for PDG-based slicing methods a criterion effectively corresponds to a pair $(s, \textsc{Vars}(s))$, where s is a statement and $\textsc{Vars}(s)$ the set of all variables *defined or used at* s.

  However, a PDG-based slicing method can compute a slice with respect to a criterion (s, V) for arbitrary V by performing the following three steps. First, the CFG node n corresponding to PDG vertex s is determined. Second, the set of CFG nodes N

corresponding to all reaching definitions for variables in V at node n are determined. Third, the set of PDG vertices S corresponding to the set of CFG nodes N is determined; the desired slice consists of all vertices from which a vertex in S can be reached.

Having dealt with these issues, we can state our conclusions concerning the accuracy of static slicing methods:

basic algorithms

For *intra*procedural static slicing, the accuracy of methods based on dataflow equations [85] (see Section 3.1.1) information-flow relations [16] (see Section 3.1.2), and PDGs [69] (see Section 3.1.3) is essentially the same, although the presentation of the computed slices differs: Weiser defines his slice to be an executable program, whereas in the other two methods, slices are defined as a subset of statements of the original program.

procedures

Weiser's *inter*procedural static slicing algorithm [85] is inaccurate for two reasons, which can be summarized as follows. First, the interprocedural summary information used to approximate the effect of a procedure call establishes relations between the set of *all* input parameters, and the set of *all* output parameters; by contrast, the approaches of [16, 44, 45, 74] determine for each output parameter the input parameters it depends upon. Second, the algorithm fails to take the call-return structure of interprocedural execution paths into account. These problems are addressed in detail in Section 3.2.1.

The algorithm by Bergeretti and Carré [16] does not compute truly interprocedural slices because only the main program is being sliced. Moreover, the it is not capable of handling recursive programs. Bergeretti-Carré slices are accurate in the sense that: (i) exact dependences between input and output parameters are used, and (ii) the calling-context problem does not occur.

The solutions of [16, 45, 44, 74] compute accurate interprocedural static slices, and are capable of handling recursive programs (see Sections 3.2.2 and 3.2.3).

arbitrary control flow

Lyle's method for computing static slices in the presence of arbitrary control flow is very conservative (see Section 3.3.1). Agrawal has shown in [2] that the solutions proposed by Gallagher and Lyle [31, 32] and by Jiang et al. are incorrect. Precise solutions for static slicing in the presence of arbitrary control flow have been proposed by Ball and Horwitz [8, 9], Choi and Ferrante [21], and Agrawal [2] (see Section 3.3.2). We conjecture that these three approaches are equally accurate.

composite variables and pointers

Lyle has presented a very conservative algorithm for static slicing in the presence of arrays (see Section 3.4). As we discussed in Section 3.4, the approach by Jiang et al. is incorrect. Agrawal et al. propose an algorithm for static slicing in the presence of arrays and pointers (see Section 3.4) that is more accurate than Lyle's algorithm.

interprocess communication

The only approach for static slicing of concurrent programs was proposed by Cheng (see Section 3.5).

3.6.3 Efficiency

Below, we will examine the efficiency of the static slicing methods that were studied earlier:

basic algorithms Weiser's algorithm for *intra*procedural static slicing based on dataflow equations [85] can determine a slice in $O(v \times n \times e)$ time[10], where v is the number of variables in the program, n the number of vertices in the CFG, and e the number of edges in the CFG.

Bergeretti and Carré report in [16] that the μ_S relation for a statement S can be computed in $O(v^2 \times n)$. From this relation, the slices for all variables at a given statement can be obtained.

Construction of a PDG essentially involves computing all data dependences and control dependences in a program. For structured programs, control dependences can be determined in a syntax-directed fashion, in $O(n)$. In the presence of arbitrary control flow, the control dependences of a single-procedure program can be computed in $O(e \times n)$ time [24, 27]. Computing data dependences essentially corresponds to determining the reaching definitions for each use. For scalar variables, this can be accomplished in $O(e \times d)$, where d is the number of definitions in the program (see, e.g., [75]). From $d \leq n$ it follows that a PDG can be constructed in $O(e \times n)$ time.

One of the self-evident advantages of PDG-based slicing methods is that, once the PDG has been computed, slices can be extracted in linear time, $O(V + E)$, where V and E are the number of vertices and edges in the *slice*, respectively. This is especially useful if several slices of the same program are required. In the worst case, when the slice consists of the entire program, V and E are equal to the number of vertices and edges of the PDG, respectively. In certain cases, there can be a quadratic blowup in the number of flow dependence edges of a PDG, e.g., $E = O(V^2)$. We are not aware of any slicing algorithms that use more efficient program representations such as the SSA form [7].

procedures In the discussion below, *Visible* denotes the maximal number of parameters and variables that are visible in the scope of any procedure, and *Params* denotes the maximum number of formal-in vertices in any procedure dependence graph of the SDG. Moreover, *TotalSites* is the total number of call sites in the program; N_p and E_p denote the number of vertices and edges in the CFG of procedure p, and $Sites_p$ the number of call sites in procedure p.

[10] In [85], Weiser states a bound of $O(n \times e \times \log(e))$. However, this is a bound on the number of "bit-vector" steps performed, where the length of each bit-vector is $O(v)$. We have multiplied the cost by $O(v)$ to account for the cost of such bit-vector operations. The problem of determining relevant variables is similar to that of determining possibly-uninitialized variables. The transformation technique of [75] can be employed to do this in $O(v \times e)$ time. At most n iterations have to be performed due to branch statements with indirect relevance. Hence, an improved bound for Weiser's intraprocedural slicing algorithm is $O(v \times n \times e)$.

Weiser does not state an estimate of the complexity of his interprocedural slicing algorithm in [85]. However, one can observe that for an initial criterion C, the set of criteria in (UP $\cup$ DOWN)*(C) contains at most $O(\mathit{Visible})$ criteria in each procedure p. An intraprocedural slice of procedure p takes time $O(\mathit{Visible} \times N_p \times E_p)$. Furthermore, computation of interprocedural summary information can be done in $O(\mathit{Globals} \times \mathit{TotalSites})$ time [23]. Therefore, the following expression constitutes an upper bound for the time required to slice the entire program:

$$O(\mathit{Globals} \times \mathit{TotalSites} + \mathit{Visible}^2 \times \Sigma_p(\mathit{Sites}_p \times N_p \times E_p))$$

The complexity of the approach by Bergeretti and Carré requires that each procedure be sliced once. Each call site is replaced by at most *Visible* assignments. Therefore, the cost of slicing procedure p is $O(\mathit{Visible}^2 \times (n + \mathit{Visible} \times \mathit{Sites}_p))$, and the total cost of computing an interprocedural slice is:

$$O(\mathit{Visible}^2 \times \Sigma_p(n + \mathit{Visible} \times \mathit{Sites}_p))$$

As was discussed in Section 3.2.1, the approach by Hwang, Du, and Chou may require time exponential in the size of the program.

Construction of the individual procedure dependence graphs of an SDG takes time $O(\Sigma_p(E_p \times N_p))$. The Horwitz-Reps-Binkley algorithm for computing summary edges takes time:

$$O(\mathit{TotalSites} \times E^{\mathrm{PDG}} \times \mathit{Params} + \mathit{TotalSites} \times \mathit{Sites}^2 \times \mathit{Params}^4)$$

where *Sites* is the maximum number of call sites in any procedure, and E^{PDG} is the maximum number of control and data dependence edges in any procedure dependence graph. (for details, the reader is referred to [44, 74]). The Reps-Horwitz-Sagiv-Rosay approach for computing summary edges requires

$$O(P \times E^{\mathrm{PDG}} \times \mathit{Params} + \mathit{TotalSites} \times \mathit{Params}^3)$$

time [74]. Here, P denotes the number of procedures in the program. Assuming that the number of procedures P is usually much less than the number of procedure calls *TotalSites*, both terms of the complexity measure of the Reps-Horwitz-Sagiv-Rosay approach are asymptotically smaller than those of the Horwitz-Reps-Binkley algorithm.

Once an SDG has been constructed, a slice can be extracted from it (in two passes) in $O(V + E)$, where V and E are the number of vertices and edges in the slice, respectively. In the worst case, $V = V^{\mathrm{SDG}}$ and $E = E^{\mathrm{SDG}}$, where V^{SDG} and E^{SDG} are the number of vertices and edges in the SDG, respectively.

arbitrary control flow Lyle's conservative algorithm for dealing with unstructured control flow is essentially the same as Weiser's algorithm [85]: a **goto** statement is included if it has a non-empty set of relevant variables. Therefore, the time requirements of Lyle's algorithm are the same as those of Weiser's: $O(v \times n \times e)$ time.

	Interprocedural Slicing	Arbitrary Control Flow	Non-scalar Variables	Interprocess Communication
D.-F. Eqs.	Weiser [85, 63]	Lyle [64]	Lyle [64]	
I.-F. Rels.	Bergeretti, Carré [16]	–	–	
PDG-based	Horwitz et al. [44] Lakhotia [59] Reps et al. [75]	Ball, Horwitz [8, 9] Choi, Ferrante [21] Agrawal [2]	Agrawal et al.[3][a]	Cheng [19]

[a] Algorithms for computing potential data dependences in the presence of non-scalar variables and aliasing can be used. See Section 3.4.

Table 3: Orthogonal problems of static slicing.

No complexity estimates are stated in [2, 9, 21]. However, the difference between these algorithms and the 'standard' PDG-based slicing algorithm is very minor: in [9, 21] a slightly different control dependence subgraph is used in conjunction with the data dependence subgraph, and in [2] the standard PDG is used in conjunction with a lexical successor tree that can be constructed in linear time, $O(n)$. Therefore it is to be expected that the efficiency of these algorithms is roughly equivalent to that of the standard, PDG-based algorithm we discussed above.

composite variables and pointers Lyle's approach for slicing in the presence of arrays [64] has the same complexity bound as Weiser's algorithm for slicing in the presence of scalar variables, because the worst-case length of reaching definitions paths remains the same.

The cost of constructing PDGs of programs with composite variables and pointers according to the algorithm proposed by Agrawal et al. in [3] is the same as that of constructing PDGs of programs with scalar variables only. This is the case because the worst-case length of (potential) reaching definitions paths remains the same, and determining maybe intersections and partial intersections (see Section 3.4) can be done in constant time.

interprocess communication Cheng doesn't state any complexity estimate for determining selection, synchronization, and communication dependence in [19]. The time required for extracting slices is $O(V + E)$, where V and E denote the number of vertices and edges in the PDN, respectively.

It should be remarked here that more accurate static slices can be determined in the presence of non-scalar variables if more advanced (but computationally expensive) data dependence analysis were performed (see, e.g., [66, 87]).

3.6.4 Combining Static Slicing Algorithms

Table 3 highlights 'orthogonal' problems of static slicing: dealing with procedures, unstructured control flow, non-scalar variables, and interprocess communication. For each computation method, the table shows which papers present a solution for these problems. In principle, solutions to different problems could be combined if they appear in the same row of Table 3 (i.e., if they apply to the same computation method).

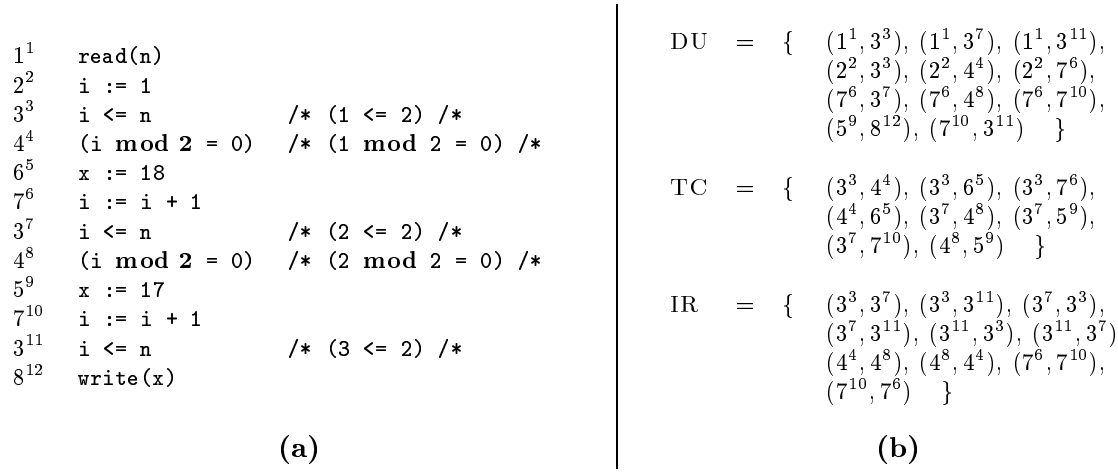


Figure 14: **(a)** Trajectory for the example program of Figure 2 **(a)**. **(b)** Dynamic Flow Concepts for this trajectory.

4 Methods for Dynamic Slicing

4.1 Basic Algorithms for Dynamic Slicing

In this section, we study basic algorithms for dynamic slicing of structured programs without nonscalar variables, procedures, and interprocess communication.

4.1.1 Dynamic Flow Concepts

Korel and Laski describe how dynamic slices can be computed in [56, 57]. They formalize the execution history of a program as a *trajectory* consisting of a sequence of 'occurrences' of statements and control predicates. Labels serve to distinguish between different occurrences of a statement in the execution history. As an example, Figure 14 shows the trajectory for the program of Figure 2 **(a)** for input `n = 2`.

A *dynamic slicing criterion* is specified as a triple (x, I^q, V) where x denotes the input of the program, statement occurrence I^q is the q^{th} element of the trajectory, and V is a subset of the variables of the program[11]. A *dynamic slice* is defined as an *executable* program that is obtained from the original program by deleting zero or more statements. For input x, the same values for variables in V are computed at 'corresponding' points in the trajectories of the program and its slice. Two further requirements are imposed on dynamic slices: (i) the statement corresponding to criterion I^q occurs in the slice, and (ii) if a loop occurs in the slice, it is traversed the same number of times as in the original program.

In order to compute dynamic slices, Korel and Laski introduce three *dynamic flow concepts* which formalize the dependences between occurrences of statements in a trajectory. The *Definition-Use* (DU) relation associates a use of a variable with its last definition.

[11]Korel and Laski's definition of a dynamic slicing criterion is somewhat inconsistent. It assumes that a trajectory is available although the input x uniquely defines this. A self-contained and minimal definition of a dynamic slicing criterion would consist of a triple (x, q, V) where q is the number of a statement occurrence in the trajectory induced by input x.

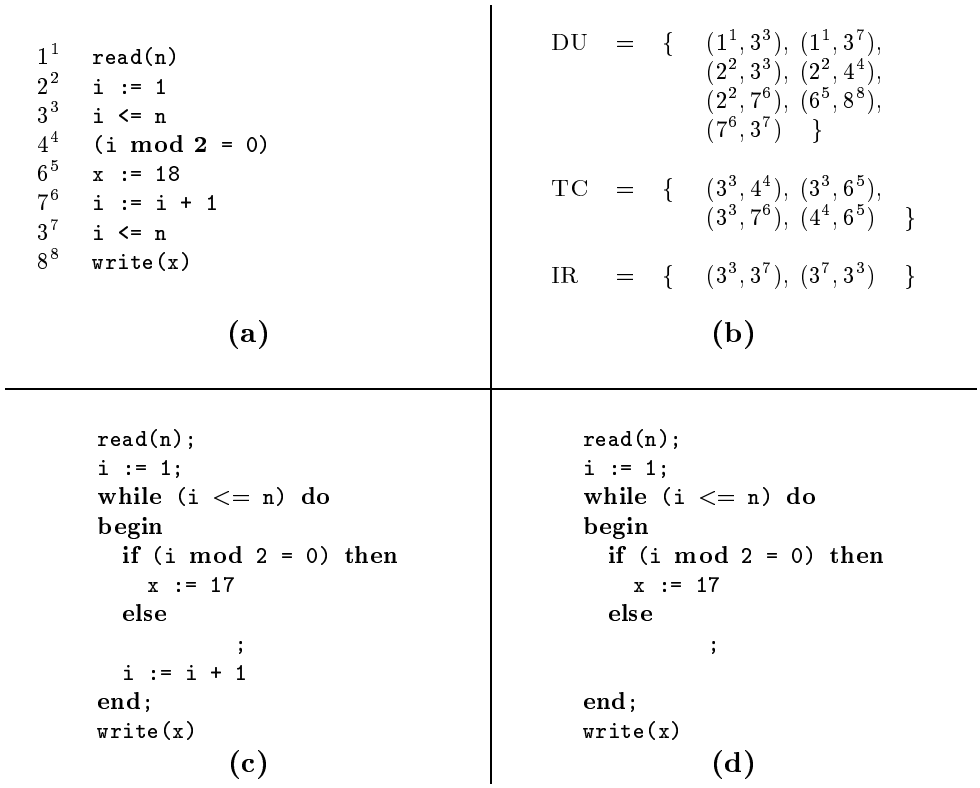


Figure 15: **(a)** Trajectory of the example program of Figure 2 **(a)**. for input n = 1. **(b)** Dynamic flow concepts for this trajectory. **(c)** Dynamic slice for criterion $(n = 1, 8^8, x)$. **(d)** Non-terminating slice obtained by ignoring the effect of the IR relation.

Note that in a trajectory, this definition is uniquely defined. The *Test-Control* (TC) relation associates the most recent occurrence of a control predicate with the statement occurrences in the trajectory that are control dependent upon it. This relation is defined in a syntax-directed manner, for structured program constructs only. Occurrences of the same statement are related by the symmetric *Identity* (IR) relation. Figure 14 **(b)** shows the dynamic flow concepts for the trajectory of Figure 14 **(a)**.

Dynamic slices are computed in an iterative way, by determining successive sets S^i of directly and indirectly relevant statements. For a slicing criterion (x, I^q, V) The initial approximation S^0 contains the last definitions of the variables in V in the trajectory, as well as the test actions in the trajectory on which I^q is control dependent. Approximation S^{i+1} is defined as follows:

$$S^{i+1} = S^i \cup A^{i+1}$$

where A^{i+1} consists of:

$$A^{i+1} = \{\, X^p \mid X^p \notin S^i,\ (X^p, Y^t) \in (\mathrm{DU} \cup \mathrm{TC} \cup \mathrm{IR}) \text{ for some } Y^t \in S^i,\ p < q \,\}$$

The dynamic slice is easily obtained from the fixpoint S_C of this process (as q is finite, this always exists): any statement X for which an instance X^p occurs in S_C will be in the slice. Furthermore, statement I corresponding to criterion I^q is added to the slice.

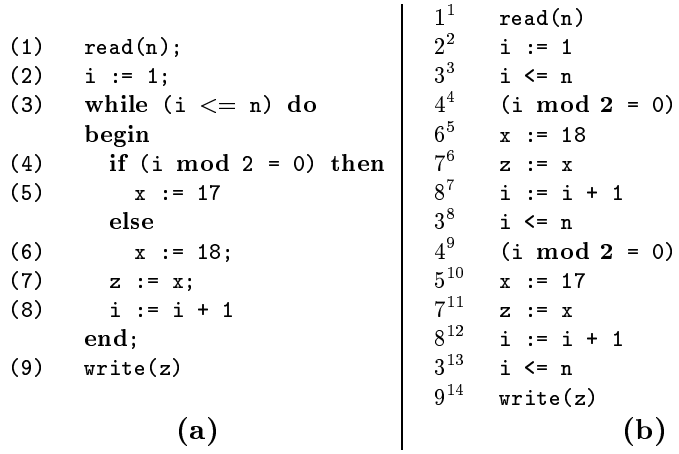


Figure 16: **(a)** Example program. **(b)** Trajectory for input n = 2.

As an example, we compute the dynamic slice for the trajectory of Figure 14 and the criterion $(n = 2, 8^{12}, \{ x \})$. Since the final statement is not control dependent on any other statement, the initial approximation of the slice consists of the last definition of x: $A^0 = \{ 5^9 \}$. Subsequent iterations will produce $A^1 = \{ 3^7, 4^8 \}$, $A^2 = \{ 7^6, 1^1, 3^3, 3^{11}, 4^4 \}$, and $A^3 = \{ 2^2, 7^{10} \}$. From this, it follows that:

$$S_C = \{ 1^1, 2^2, 3^3, 4^4, 7^6, 3^7, 4^8, 5^9, 7^{10}, 3^{11}, 8^{12} \}$$

Thus, the dynamic slice with respect to criterion $(n = 2, 8^{12}, \{x\})$ includes every statement except statement 5, corresponding to statement 6^5 in the trajectory. This slice was shown earlier in Figure 2 **(b)**.

The role of the IR relation calls for some clarification. To this end, we consider the trajectory of the example program of Figure 2 **(a)** for input n = 1, which is shown in Figure 15 **(a)**. The dynamic flow concepts for this trajectory, and the slice with respect to criterion $(n = 1, 8^8, \{ x \})$ are shown in Figure 15 **(b)** and **(c)**, respectively. Note that the slice thus obtained is a terminating program. However, if we would compute the slice without taking the IR relation into account, the non-terminating program of Figure 15 **(d)** would be obtained. The reason for this phenomenon (and thus for introducing the IR relation) is that the DU and TC relations only traverse the trajectory in the *backward* direction. The purpose of the IR relation is to traverse the trajectory in *both* directions, and to include all statements and control predicates that are necessary to ensure termination of loops in the slice. Unfortunately, no proof is provided that this is always sufficient.

Unfortunately, traversing the IR relation in the 'backward' direction causes inclusion of statements that are not necessary to preserve termination. For example, Figure 16 **(a)** shows a slightly modified version of the program of Figure 2 **(a)**. Figure 16 **(b)** shows the trajectory for this program. From this trajectory, it follows that $(7^6, 7^{11}) \in$ IR, $(6^5, 7^6) \in$ DU, and $(5^{10}, 7^{11}) \in$ DU. Therefore, both statements (5) and (6) will be included in the slice, although statement (6) is neither needed to compute the final value of `z` nor to preserve termination.

It would be interesting to investigate if using a dynamic variation of Podgurski and Clarke's notion of weak control dependence [70] instead of the IR relation would lead to more accurate slices.

$$
\begin{array}{lcl}
\overline{\lambda}_{\epsilon} & = & \emptyset \\
\overline{\mu}_{\epsilon} & = & \emptyset \\
\overline{\rho}_{\epsilon} & = & \textsc{Id}
\end{array}
$$

$$
\begin{array}{lcl}
\overline{\lambda}_{v:=e} & = & \textsc{Vars}(e) \times \{\, e \,\} \\
\overline{\mu}_{v:=e} & = & \{\, (e, v) \,\} \\
\overline{\rho}_{v:=e} & = & (\textsc{Vars}(e) \times \{\, v \,\}) \;\cup\; (\textsc{Id} - (v, v))
\end{array}
$$

$$
\begin{array}{lcl}
\overline{\lambda}_{S_1;S_2} & = & \overline{\lambda}_{S_1} \;\cup\; \overline{\rho}_{S_1} \cdot \overline{\lambda}_{S_2} \\
\overline{\mu}_{S_1;S_2} & = & \overline{\mu}_{S_1} \cdot \overline{\rho}_{S_2} \;\cup\; \overline{\mu}_{S_2} \\
\overline{\rho}_{S_1;S_2} & = & \overline{\rho}_{S_1} \cdot \overline{\rho}_{S_2}
\end{array}
$$

$$
\overline{\lambda}_{\textbf{if } e \textbf{ then } s} = \begin{cases} (\textsc{Vars}(e) \times \{\, e \,\}) \;\cup\; \overline{\lambda}_S & \text{if } e \text{ evaluates to true} \\ \emptyset & \text{if } e \text{ evaluates to false} \end{cases}
$$

$$
\overline{\mu}_{\textbf{if } e \textbf{ then } s} = \begin{cases} (\{\, e \,\} \times \textsc{Defs}(S)) \;\cup\; \overline{\mu}_S & \text{if } e \text{ evaluates to true} \\ \emptyset & \text{if } e \text{ evaluates to false} \end{cases}
$$

$$
\overline{\rho}_{\textbf{if } e \textbf{ then } s} = \begin{cases} (\textsc{Vars}(e) \times \textsc{Defs}(S)) \;\cup\; \overline{\rho}_S & \text{if } e \text{ evaluates to true} \\ \textsc{Id} & \text{if } e \text{ evaluates to false} \end{cases}
$$

$$
\overline{\lambda}_{\textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2} = \begin{cases} (\textsc{Vars}(e) \times \{\, e \,\}) \;\cup\; \overline{\lambda}_{S_1} & \text{if } e \text{ evaluates to true} \\ (\textsc{Vars}(e) \times \{\, e \,\}) \;\cup\; \overline{\lambda}_{S_2} & \text{if } e \text{ evaluates to false} \end{cases}
$$

$$
\overline{\mu}_{\textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2} = \begin{cases} (\{\, e \,\} \times \textsc{Defs}(S_1)) \;\cup\; \overline{\mu}_{S_1} & \text{if } e \text{ evaluates to true} \\ (\{\, e \,\} \times \textsc{Defs}(S_2)) \;\cup\; \overline{\mu}_{S_2} & \text{if } e \text{ evaluates to false} \end{cases}
$$

$$
\overline{\rho}_{\textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2} = \begin{cases} (\textsc{Vars}(e) \times \textsc{Defs}(S_1)) \;\cup\; \overline{\rho}_{S_1} & \text{if } e \text{ evaluates to true} \\ (\textsc{Vars}(e) \times \textsc{Defs}(S_2)) \;\cup\; \overline{\rho}_{S_2} & \text{if } e \text{ evaluates to false} \end{cases}
$$

Figure 17: Definition of dynamic dependence relations.

4.1.2 Dynamic Dependence Relations

Gopal describes an approach were *dynamic dependence relations* are used to compute dynamic slices in [33]. He introduces dynamic versions of Bergeretti and Carré's information-flow relations[12] λ_S, μ_S, and ρ_S (see Section 3.1.2). The $\overline{\lambda}_S$ relation contains all pairs (v, e) such that statement e depends on the input value of v when program S is executed. Relation $\overline{\mu}_S$ contains all pairs (e, v) such that the output value of v depends on the execution of statement e. A pair (v, v') is in relation $\overline{\rho}_S$ if the output value of v' depends on the input value of v. In these definitions, it is presumed that S is executed for some fixed input.

For empty statements, assignments, and statement sequences Gopal's dependence relations are exactly the same as for the static case. The (static) information-flow relations for a conditional statement are derived from the statement itself, and from the statements that constitute its branches. For dynamic dependence relations, however, only the dependences that arise in the branch that is actually executed are taken into account. As in [16], the dependence relation for a **while** statement (omitted here) is expressed in terms of dependence relations for nested conditionals with equivalent behavior. However, whereas

[12] Gopal uses the notation s_v^S, v_v^S, and v_s^S. In order to avoid confusion and to make the relation with Bergeretti and Carré's work explicit (see Section 3.1.2), we will use $\overline{\lambda}_S$, $\overline{\mu}_S$, and $\overline{\rho}_S$ instead.

Expression #[a]	Affected Variables
1	{ i, n, x }
2	{ i, x }
3	{ i, x }
4	{ i, x }
5	{ x }
6	∅
7	{ i, x }
8	∅

[a] Expressions are indicated by the line numbers in Figure 2.

Figure 18: The $\overline{\mu}$ relation for the example program of Figure 2 **(a)** and input `n = 2`.

in the static case loops are effectively replaced by their infinite unwindings, the dynamic case only requires that a loop be unwound k times, where k is the number of times the loop executes. The resulting definitions are very convoluted because the dependence relations for the body of the loop may differ in each iteration. Hence, a simple transitive closure operation, as was used in the static case, is insufficient.

Figure 17 summarizes Gopal's dynamic dependence relations. Here, Defs(S) denotes the set of variables that is modified by executing statement S. Using these relations, a dynamic slice w.r.t. the final value of a variable v is defined as:

$$\sigma_v^P \equiv \{e \mid (e, v) \in \overline{\mu}_P\}$$

Figure 18 **(a)** shows the information-flow relation $\overline{\mu}$ for the (entire) program of Figure 2 **(a)**[13]. From this relation it follows that the set of expressions which for input `n = 2` affect the value of `x` at the end of the program are { 1, 2, 3, 4, 5, 7 }. The corresponding dynamic slice is nearly identical to the slice shown in Figure 1 **(b)**, the only difference being the fact that Gopal's algorithm also excludes the final statement `write(x)` on line 8.

For the previous example, Gopal's dependence relations computed a similar slice to that computed in Section 4.1.1; the only difference being the fact that the former omitted the `write(x)` statement. However, for certain cases, Gopal's algorithm may compute a non-terminating slice of a terminating program. Figure 19 **(a)** shows the slice for the program of Figure 2 and input `n = 1` as computed according to Gopal's algorithm.

An advantage of using dependence relations is that, for certain cases, smaller slices are computed than by Korel and Laski's algorithm. For example, Figure 19 **(b)** shows the slice with respect to the final value of `z` for the example program of Figure 16 **(a)**, for input `n = 2`. Observe that the assignment `x := 18`, which occurs in the slice computed by the algorithm of Section 4.1.1, is not included in the slice here.

4.1.3 Dependence Graphs

Miller and Choi were the first to introduce a dynamic variation of the PDG, called the *dynamic program dependence graph* in [67]. These graphs are used by their parallel program debugger to perform *flowback analysis* [10] and constructed incrementally, on demand.

[13] Gopal does not define information-flow relations for I/O statements. For the purposes of this example, it is assumed that the statement `read(n)` can be treated as an assignment `n :=` *SomeConstant*, and that the statements `write(sum)` and `write(product)` should be treated as empty statements.

```
read(n);
i := 1;
while (i <= n) do
begin
  if (i mod 2 = 0) then

  else
    x := 18;

end;
```

(a)

```
read(n);
i := 1;
while (i <= n) do
begin
  if (i mod 2 = 0) then
    x := 17
  else
          ;
  z := x;
  i := i + 1
end;
```

(b)

Figure 19: **(a)** Non-terminating slice computed for example program of Figure 2 **(a)** with respect to the final value of `x`, for input `n = 1`. **(b)** Slice for the example program of Figure 16 **(a)** with respect to the final value of `x`, for input `n = 2`.

Prior to execution, a (variation of a) static PDG is constructed, and the object code of the program is augmented with code which generates a log file. In addition, an emulation package is generated. Programs are partitioned into so-called emulation blocks (typically, a subroutine). During debugging, the debugger uses the log file, the static PDG, and the emulation package to re-execute an emulation block, and obtain the information necessary to construct the part of the dynamic PDG corresponding to that block. In case the user wants to perform flowback analysis to parts of the graph that have not been constructed yet, more emulation blocks are re-executed.

In [5], Agrawal and Horgan develop an approach for using dependence graphs to compute dynamic slices. Their first two algorithms for computing dynamic slices are inaccurate, but useful for understanding their final approach. The initial approach uses the PDG as it was discussed in Section 3.1.3[14], and marks the *vertices* that are executed for a given test set. A dynamic slice is computed by computing a static slice in the subgraph of the PDG that is induced by the marked vertices. By construction, this slice only contains vertices that were executed. This solution is imprecise because it does not detect situations where there exists a flow edge in the PDG between a marked vertex v_1 and a marked vertex v_2, but where the definitions of v_1 are not actually used at v_2.

For example, Figure 21 **(a)** shows the PDG of the example program of Figure 2 **(a)**. Suppose we want to compute the slice w.r.t. the final value of `x` for input `n = 2`. All vertices of the PDG are executed, causing all PDG vertices to be marked. The static slicing algorithm of Section 3.1.3 will therefore produce the entire program as the slice, even though the assignment `x := 18` is irrelevant. This assignment is included in the slice because there exists a dependence edge from vertex `x := 18` to vertex `write(x)` even though this edge does not represent a dependence that occurs during the second iteration of the loop. More precisely, this dependence only occurs in iterations of the loop where the control variable `i` has an odd value.

The second approach consists of marking PDG *edges* as the corresponding dependences

[14]The dependence graphs of [5] do not have an entry vertex. The absence of an entry vertex does not result in a different slice. For reasons of uniformity, all dependence graphs shown in the present paper have an entry vertex.

arise during execution. Again, the slice is obtained by traversing the PDG, but this time only along marked edges. Unfortunately, this approach still produces imprecise slices in the presence of loops because an edge that is marked in some loop iteration will be present in all subsequent iterations, even when the same dependence does not recur. Figure 21 **(b)** shows the PDG of the example program of Figure 16 **(a)**. For input `n = 2`, all dependence edges will be marked, causing the slice to consist of the entire program. It is shown in [5] that a potential refinement of the second approach consisting of *un*marking edges of previous iterations is incorrect.

Agrawal and Horgan point out the interesting fact that their second approach for computing dynamic slices produces identical results as the algorithm proposed by Korel and Laski (see Section 4.1.1). However, the PDG of a program (with optionally marked edges) requires only $O(n^2)$ space (n denotes the number of statements in the program), whereas Korel and Laski's trajectories are in principle unbounded in size.

Agrawal and Horgan's second approach computes imprecise slices because it does not account for the fact that different occurrences of a statement in the execution history may be (transitively) dependent on different statements. This observation motivates their third solution: create a distinct vertex in the dependence graph for each occurrence of a statement in the execution history. This kind of graph is referred to as a *Dynamic Dependence Graph* (DDG). A dynamic slicing criterion is identified with a vertex in the DDG, and a dynamic slice is computed by determining all DDG vertices from which the criterion can be reached. A statement or control predicate is included in the slice if the criterion can be reached from at least one of the vertices for its occurrences.

Figure 21 shows the DDG for the example program of Figure 2 **(a)**. The slicing criterion corresponds to the vertex labeled `write(z)`, and all vertices from which this vertex can be reached are indicated by shading. Observe that the criterion cannot be reached from the vertex labeled `x := 18`. Therefore the corresponding assignment is not in the slice.

The disadvantage of using DDGs is that the number of vertices in a DDG is equal to the number of executed statements, which is unbounded. The number of dynamic slices, however, is in the worst case $O(2^n)$, where n is the number of statements in the program being sliced. Figure 20 shows a program Q^n that has $O(2^n)$ dynamic slices. The program reads a number of values in variables x_i $(1 \leq i \leq n)$, and allows one to compute the sum $\sum_{x \in S} x$, for any number of subsets $S \subseteq \{ x_1, \cdots, x_n \}$. The crucial observation here is that, in each iteration of the outer loop, the slice with respect to statement `write(y)` will contain exactly the statements `read(`x_i`)` for $x_i \in S$. Since a set with n elements has 2^n subsets, program Q^n has $O(2^n)$ different dynamic slices.

Agrawal and Horgan propose to reduce the number of vertices in the DDG by merging vertices for which the transitive dependences map to the same set of statements. In other words, a new vertex is only introduced if it can create a new dynamic slice. Obviously, this check involves some run-time overhead. The resulting graph is referred to as the *Reduced Dynamic Dependence Graph* (RDDG) of a program. Slices computed using RDDGs have the same precision as those computed using DDGs.

In the DDG of Figure 21 **(c)**, the vertices labeled `i := i + 1` and the rightmost two vertices labeled `i <= n` have the same transitive dependences; these vertices depend on statements 1, 2 ,3, and 8 of the program of Figure 16 **(a)**. Hence, the RDDG for this program, and input `n = 2` is obtained by merging these four DDG vertices into one vertex.

In [5], an algorithm is presented for the construction of an RDDG without having to

```
program Q^n;
  read(x_1);
  ...
  read(x_n);
  MoreSubsets := true;
  while MoreSubsets do
  begin
    Finished := false;
    y := 0;
    while not(Finished) do
    begin
      read(i);
      case (i) of
        1:  y := y + x_i;
        ...
        n:  y := y + x_n;
      end;
      read(Finished);
    end;
    write(y);
    read(MoreSubsets);
  end
end.
```

Figure 20: Program Q^n with $O(2^n)$ different dynamic slices.

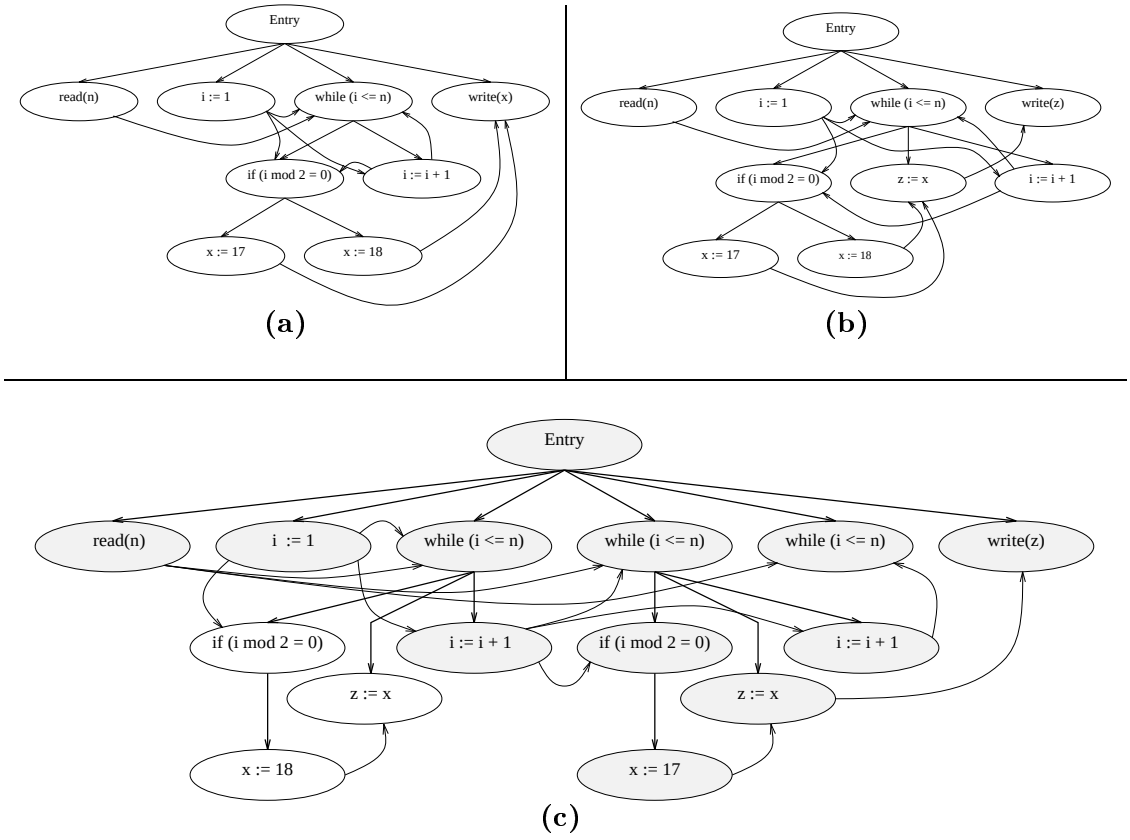


Figure 21: **(a)** PDG of the program of Figure 2 **(a)**. **(b)** PDG of the program of Figure 16 **(a)**. **(c)** DDG of the program of Figure 16 **(a)**.

keep track of the entire execution history. The information that needs to be maintained is: (i) for each variable, the vertex corresponding to its last definition, (ii) for each predicate, the vertex corresponding to its last execution, and (iii) for each vertex in the RDDG, the dynamic slice w.r.t. that vertex.

4.2 Interprocedural Dynamic Slicing

In [3], Agrawal, DeMillo and Spafford consider dynamic slicing of procedures with various parameter-passing mechanisms. In the discussion below, it is assumed that a procedure P with formal parameters $f_1, \cdots, f_n$ is called with actual parameters $a_1, \cdots, a_n$. It is important to realize that in the approach of [3], dynamic data dependences based on definitions and uses of *memory locations* are used. This way, two potential problems are avoided. First, the use of global variables inside procedures does not pose any problems. Second, no alias analysis is required.

Call-by-value parameter-passing is modeled by a sequence of assignments $f_1{:=}a_1; \cdots; f_n{:=}a_n$ which is executed before the procedure is entered. In order to determine the memory cells for the correct activation record, the USE sets for the actual parameters a_i are determined before the procedure is entered, and the DEF sets for the formal parameters f_i after the procedure is entered. For *Call-by-value-result* parameter-passing, additional assignments of formal parameters to actual parameters have to be performed upon exit from the procedure. *Call-by-reference* parameter-passing does not require any actions specific to dynamic slicing, as the same memory cell is associated with corresponding actual and formal parameters a_i and f_i.

An alternative approach for interprocedural dynamic slicing was presented by Kamkar, Shahmehri, and Fritzson in [52, 51]. This work distinguishes itself from the solution by Agrawal et al. by the fact that the authors are primarily concerned with procedure-level slices. That is, they study the problem of determining the set of *call sites* in a program that affect the value of a variable at a particular call site.

During execution, a *(dynamic dependence) summary graph* is constructed. The vertices of this graph, referred to as *procedure instances*, correspond to procedure activations annotated with their parameters[15]. The edges of the summary graph are either activation edges corresponding to procedure calls, or summary dependence edges. The latter type reflects transitive data and control dependences between input and output parameters of procedure instances.

A *slicing criterion* is defined as a pair consisting of a procedure instance, and an input or output parameter of the associated procedure. After constructing the summary graph, a slice with respect to a slicing criterion is determined in two steps. First, the parts of the summary graph from which the criterion can be reached is determined; this subgraph is referred to as an *execution slice*. Vertices of an execution slice are *partial* procedure instances, because some parameters may be 'sliced away'. An interprocedural program slice consists of all call sites in the program for which a partial instance occurs in the execution slice.

Three approaches for constructing summary graphs are considered. In the first approach, *intra*procedural data dependences are determined statically which may result in

[15]More precisely, Kamkar refers to the *incoming* and *outgoing* variables of a procedure. This notion also applies to global variables which are referenced or modified in a procedure.

inaccurate slices in the presence of conditionals. In the second approach, all dependences are determined at run-time. While this results in accurate dynamic slices, the dependences for a procedure P have to be re-computed every time P is called. The third approach attempts to combine the efficiency of the 'static' approach with the accuracy of the 'dynamic' approach by computing the dependences inside basic blocks statically, and the inter-block dependences dynamically. In all approaches control dependences[16] are determined statically. It is unclear how useful this third approach is in the presence of composite variables and pointers, where the run-time intra-block dependences cannot be determined statically: additional alias analysis would have to be performed at run-time.

In [49], Kamkar adapts the interprocedural slicing method of [51, 52] to compute statement-level interprocedural slices (i.e., slices consisting of a set of statements instead of a set of call sites). In essence, this is accomplished by introducing a vertex for each *statement instance* (instead of each procedure instance) in the summary graph. The same three approaches (static, dynamic, combined static/dynamic) for constructing summary graphs can be used.

Choi, Miller and Netzer discuss an approach for interprocedural flowback analysis in [22]. Initially, it is assumed that a procedure call may modify every global variable; to this end, the static PDG is augmented with *linking* edges indicating *potential* data dependences. In a second phase, interprocedural summary information is used to either replace linking edges by data dependence edges, or delete them from the graph. Some linking edges may remain; these have to be resolved at run-time.

4.3 Dynamic Slicing in the Presence of Composite Datatypes/Pointers

4.3.1 Dynamic Flow Concepts

In [57], Korel and Laski consider slicing in the presence of composite variables by regarding each element of an array, or field of a record as a distinct variable. Dynamic data structures are treated as *two* distinct entities, namely the pointer itself and the object being pointed to. For dynamically allocated objects, they propose a solution where a unique name is assigned to each object.

4.3.2 Dependence Graphs

Agrawal, DeMillo, and Spafford present a dependence graph based algorithm for dynamic slicing in the presence of composite datatypes and pointers in [3]. Their solution consist of expressing DEF and USE sets in terms of *actual memory locations* provided by the compiler. The algorithm of [3] is similar to that for *static* slicing in the presence of composite datatypes and pointers by the same authors (see Section 3.4). However, during the computation of *dynamic* reaching definitions, no *Maybe* intersections can occur—only *Complete* and *Partial* intersections.

Choi, Miller, Netzer extend the flowback analysis method of [67] (see Section 4.1.3) in order to deal with arrays and pointers. For arrays, *linking edges* are added to their static PDGs; these edges express *potential* data dependences which are either deleted or changed

[16] Kamkar et al. use a notion of *termination-preserving* control dependence that is similar to Podgurski and Clarke's *weak* control dependence [70].

into genuine data dependences at run-time. Pointer accesses are resolved at run-time, by recording all uses of pointers in the log file.

4.4 Dynamic Slicing of Distributed Programs

4.4.1 Dynamic Flow Concepts

Korel and Ferguson extend the dynamic slicing method of [56, 57] to distributed programs with Ada-type rendezvous communication (see, e.g., [12]). For a distributed program, the execution history is formalized as a *distributed program path* which, for each task, comprises of: (i) the sequence of statements (trajectory) executed by it, and (ii) a sequence of triples (A, C, B) identifying each rendezvous the task is involved in. Here, A identifies the **accept** statement in the task, B identifies the other task that participated in the communication, and C denotes the entry call statement in the task that was involved in the rendezvous.

A dynamic slicing criterion of a distributed program specifies: (i) the input of each task, (ii) a distributed program path P, (iii) a task w, (iv) a statement occurrence q in the trajectory of w, and (v) a variable v. A dynamic slice with respect to such a criterion is an executable projection of the program that is obtained by deleting statements from it. However, the program is only guaranteed to preserve the behavior of the program if the rendezvous in the slice occur in the same relative order as in the program. (Note that not all rendezvous of the program need to be in the slice.)

The method for computing slices of distributed programs of [55] is basically a generalization of the method of [56, 57], though stated in a slightly different manner. In addition to the previously discussed dynamic flow concepts (see Section 4.1.1), a notion of *communication influence* is introduced, to capture the interdependences between tasks. The authors also present a distributed version of their algorithm which uses a separate process for slicing each task.

4.4.2 Dependence Graphs

Duesterwald, Gupta, and Soffa present a dependence graph based algorithm for computing dynamic slices of distributed programs [25]. They introduce a Distributed Dependence Graph (DDG)[17] for representing distributed programs.

A distributed program P consists of a set of processes $P_1, \cdots, P_n$. Communication between processes is assumed to be synchroneous and nondeterministic and is expressed by way of **send** and **receive** statements. A *distributed dynamic slicing criterion* $(I_1, X_1), \cdots, (I_n, X_n)$ specifies for each process P_i its input I_i, and a set of statements X_i. A *distributed dynamic slice* S is an executable set of processes $P'_1, \cdots, P'_n$ such that the statements of P'_i are a subset of those of P_i. Slice S computes the same values at statements in each X_i as program P does, when executed with the same input. This is accomplished by: (i) including *all* input statements in the slice, and (ii) replacing nondeterministic communication statements in the program by deterministic communication statements in the slice.

A DDG contains a single vertex for each statement and control predicate in the program. Control dependences between statements are determined statically, prior to execution. Edges for data and communication dependences are added to the graph at run-time.

[17] This abbreviation "DDG" used in Section 4.4.2 should not be confused with the notion of a Dynamic Dependence Graph that was discussed earlier in Section 4.1.

Slices are computed in the usual way by determining the set of DDG vertices from which the vertices specified in the criterion can be reached. Both the construction of the DDG and the computation of slices is performed in a distributed manner; a separate DDG construction process and slicing process is assigned to each process P_i in the program; these processes communicate when a **send** or **receive** statement is encountered.

Due to the fact that a single vertex is used for all occurrences of a statement in the execution history, inaccurate slices may be computed in the presence of loops (see Section 4.1.1). For example, the slice with respect to the final value of `z` for the program of Figure 16 with input `n = 2` will be the entire program.

Cheng presents an alternative dependence graph based algorithm for computing dynamic slices of distributed and concurrent programs in [19]. The PDN representation of a concurrent program (see Section 3.5) is used for computing dynamic slices. Cheng's algorithm is basically a generalization of the initial approach proposed by Agrawal and Horgan in [5]: the PDN vertices corresponding to executed statements are marked, and the static slicing algorithm of Section 3.5 is applied to the PDN subgraph induced by the marked vertices. As was discussed in Section 4.1.3, this yields inaccurate slices.

In [22, 67], Choi et al. describe how their approach for flowback analysis can be extended to parallel programs. Shared variables with semaphores, message-passing communication, and Ada-type rendezvous mechanisms are considered. To this end, a parallel dynamic graph is introduced which contains synchronization vertices for synchronization operations (such as P and V on a semaphore) and synchronization edges which represent dependences between concurrent processes. Choi et al. explain how, by analysis of the parallel dynamic graph, read/write and write/write conflicts between concurrent processes can be found.

4.5 Comparing Methods for Dynamic Slicing

In this section, we compare and classify the dynamic slicing methods that were presented earlier. The section is organized as follows: Section 4.5.1 summarizes the problems that are addressed in the literature. Sections 4.5.2 and 4.5.3 compare the *accuracy* and *efficiency* of slicing methods that address the same problem, respectively. Finally, Section 4.5.4 investigates the possibilities for 'combining' algorithms that deal with different problems.

4.5.1 Overview

Table 4 lists the dynamic slicing algorithms that we discussed earlier, and summarizes the issues studied in each paper. For each paper, the table shows: (i) the computation method, (ii) whether or not the computed slices are executable programs, (iii) whether or not an interprocedural solution is supplied, (iv) the data types under consideration, and (v) whether or not interprocess communication is considered. Similar to Table 2, the table only shows problems that *have been addressed*. It does *not* indicate how various algorithms may be combined, and it also does not give an indication of the quality of the work.

Unlike in the static case, there exists a significant difference between methods which compute executable slices [25, 55, 56, 57], and approaches which compute slices that are merely sets of statements [3, 5, 33]. The latter type of slice may not be executable due

	Computation Method[a]	Executable	Interprocedural Solution	Data Types[b]	Interprocess Communication
Korel, Laski [56, 57]	D	yes	no	S, A, P	no
Korel, Ferguson [55]	D	yes	no	S, A	yes
Gopal [33]	I	no	no	S	no
Agrawal, Horgan [5]	G	no	no	S	no
Agrawal et al. [1, 3]	G	no	yes	S, A, P	no
Kamkar et al. [51, 52]	G	no	yes	S	no
Duesterwald et al. [25]	G	yes	no	S, A, P	yes
Cheng [19]	G	no	no	S	yes
Choi et al. [22, 67]	G	no	yes	S, A, P	yes

[a] D = dynamic flow concepts, I = dynamic dependence relations, G = reachability in a dependence graph.

[b] S = scalar variables, A = arrays/records, P = pointers.

Table 4: Overview of dynamic slicing methods.

to the absence of assignments for incrementing loop counters[18]. For convenience, we will henceforth refer to such slices as 'non-executable' slices. As we discussed in Section 4.1.1, the algorithms that compute executable dynamic slices may produce inaccurate results in the presence of loops.

Apart from the work by Venkatesh [79], there is very little semantic justification for any of the methods for computing 'non-executable' slices. The algorithms of [5, 19, 51, 52, 67] are graph-reachability algorithms that compute a set of statements that directly or indirectly 'affect' the values computed at the criterion. Besides the algorithms themselves, little or no attention is paid to formal characterization of such slices.

4.5.2 Accuracy

basic algorithms The slices computed by Korel and Laski's algorithm [56, 57] (see Section 4.1.1) are less accurate than those computed by the algorithms by Agrawal and Horgan [5] (see Section 4.1.3) and Gopal [33] (see Section 4.1.2). This is due to Korel and Laski's constraint that their slices should be executable. Slices of terminating programs, as computed by Agrawal and Horgan and Gopal, may consist of diverging programs.

procedures Dependence graph based algorithms for interprocedural dynamic slicing were proposed by Agrawal, DeMillo, and Spafford [3], and by Kamkar et al. [51, 52] (see Section 4.2). It is unclear if one of these algorithms procedures more accurate slices than the other.

composite variables and pointers Korel and Laski [57] (see Section 4.1.1), and Agrawal, DeMillo, and Spafford (see Section 4.1.3) proposed methods for dynamic slicing in the presence of composite variables and pointers. We are unaware of any difference in accuracy.

interprocess communication Korel and Ferguson [55] (see Section 4.4.1) and Duesterwald, Gupta, and Soffa [25] (see Section 4.4.2) compute executable slices, but deal

[18] Of course, such a slice may be executed anyway; however, it may not terminate.

with nondeterminism in a different way: the former approach requires a mechanism for replaying rendezvous in the slice in the same relative order as they appeared in the original program, whereas the latter approach replaces nondeterministic communication statements in the program by deterministic communication statements in the slice. Cheng [19] and Choi et al. [22, 67] (see Section 4.4.2) do not address this problem because the slices they compute are not necessarily executable. The methods by Cheng and Duesterwald et al. are inaccurate because static dependence graphs are used for computing dynamic slices (see the discussion in Section 4.1.3).

4.5.3 Efficiency

Since dynamic slicing involves run-time information, it is not surprising that all dynamic slicing methods discussed in this section have time requirements that depend on the number of executed statements (or procedure calls in the case of [51, 52]) N. All algorithms spend at least $O(N)$ time during execution in order to store the execution history of the program, or to update dependence graphs. Certain algorithms (e.g., [55, 56, 57]) traverse the execution history in order to extract the slice and thus require again *at least* $O(N)$ time *for each slice*, whereas other algorithms require less (sometime even constant) time. When we discuss time requirements in the discussion below, we will ignore the time spent during execution that is needed to construct histories or dependence graphs. Space requirements will always be discussed in detail.

basic algorithms Korel and Laski's solution [56, 57] (see Section 4.1.1) requires $O(N)$ space to store the trajectory, and $O(N^2)$ space to store the dynamic flow concepts. Construction of the flow concepts requires $O(N \times (v + n))$ time, where v and n are the number of variables and statements in the program, respectively. Extracting a single slice from the computed flow concepts can be done in $O(N)$ time.

The algorithm by Gopal [33] (see Section 4.1.2) requires $O(N)$ space to store the execution history and $O(n \times v)$ space to store the $\overline{\mu}_S$ relation. The time required to compute the $\overline{\mu}_S$ relation for a program S is bounded by $O(N^2 \times v^2)$. From this relation, slices can be extracted in $O(v)$ time.

As we discussed in Section 4.1.3, the slicing method proposed by Agrawal and Horgan requires at most $O(2^n)$ space, where n is the number of statements in the program. Since vertices in an RDDG are annotated with their slice during execution, slices can be extracted from it in $O(1)$.

procedures The interprocedural dynamic slicing method proposed by Kamkar et al. [51, 52] (see Section 4.2) requires $O(P^2)$ space to store the summary graph, where P is the number of executed procedure calls. A traversal of this graph is needed to extract a slice; this takes $O(P^2)$ time.

The time and space requirements of the method by Agrawal, DeMillo, and Spafford [3] are essentially the same as those of the Agrawal-Horgan basic slicing method we discussed above.

composite variables and pointers The algorithms by Korel and Laski [57] (see Section 4.3.1) and Agrawal, DeMillo, and Spafford [3] (see Section 4.3.2) for slicing in

	Interprocedural Slicing	Composite Vars/ Pointers	Interprocess Communication
Dyn. Flow Concepts	-	Korel, Laski [56, 57]	Korel, Ferguson [55]
Dyn. Dep. Relations	Gopal [33]	-	-
Dependence Graphs	Agrawal et al. [3] Kamkar et al. [51, 52]	Agrawal et al. [3]	Duesterwald et al. [25] Cheng [19] Choi et al. [22, 67]

Table 5: Orthogonal problems of dynamic slicing.

the presence of composite variables and pointers are adaptations of the basic slicing algorithms by Korel and Laski and Agrawal and Horgan, respectively (see the discussion above). These adaptations, which essentially consist of a change in the reaching definitions functions that is used to determine data dependences, does not affect the worst-case behavior of the algorithms. Therefore, we expect the time and space requirements to be the same as in the scalar variable case.

interprocess communication The algorithms by Cheng [19] and Duesterwald et al. [25] are based on *static* PDGs. Therefore, only $O(n^2)$ space is required to store the dependence graph, and slices can be extracted in $O(n^2)$ time. The distributed slicing algorithm in [25] uses a separate slicing process for each process in the program; the slicing process for process P_i requires time $O(e_i)$, where e_i is the number of edges in the PDG for process P_i. The communication overhead between the slicing processes requires at most $O(e)$ time, where e is the number of edges in the entire graph.

4.5.4 Combining Dynamic Slicing Algorithms

Table 5 displays solutions to 'orthogonal' problems of dynamic slicing: dealing with procedures, composite variables and pointers, and communication between processes. The algorithms based on dynamic flow concepts for dealing with composite variables/pointers [57], and interprocess communication [55] may be integrated with little problems. For dependence graphs, however, the situation is slightly more complicated because:

- Different graph representations are used. Agrawal et al. [3], Kamkar et al. [51, 52] and Choi et al. [22, 67] use dynamic dependence graphs with distinct vertices for different occurrence of statements in the execution history. In contrast, Duesterwald et al. [25] and Cheng [19] use variations of static PDGs.

- The dynamic slicing by Agrawal et al. [3] is based on definition and use of memory locations. All other dependence graph based slicing methods are based on definitions and uses of variable names.

Furthermore, it is unclear if the combined static/dynamic interprocedural slicing approach by Kamkar et al. [51, 52] is practical in the presence of composite variables and pointers, because the intra-block dependences cannot be determined statically in this case, and additional alias analysis would be required at run-time.

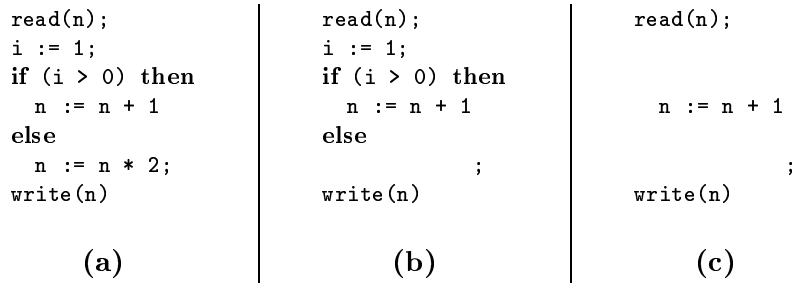


Figure 22: **(a)** Example program. **(b)** Accurate slice obtained by performing constant propagation. **(c)** Minimal slice.

5 More Accurate Slicing

5.1 Language-specific and Syntactic Issues

Although most slicing algorithms are stated in a language-independent way, some language-specific and syntactic issues cannot be avoided in practice [46]. In [82], Weiser states that "good source language slicing requires transformations beyond statement deletion". This is for example the case when a language does not allow **if** statements with empty branches, and where a slicing algorithm would exclude all statements in one of its branches. In fact, two characteristics of all slicing methods discussed so far are:

- Slices are obtained by deleting statements from a program.
- Slices are computed by tracing data and control dependences backwards from the slicing criterion.

However, if the singular objective is to obtain slices that are as small as possible, both of these constraints need to be dismissed.

Consider for example the program of Figure 22 **(a)**. When asked for the slice with respect to statement `write(n)`, traditional slicing algorithms will produce the entire program. However, by using constant propagation techniques [80], one can determine that the value of `i` is constant, causing the **else** branch of the conditional never to be selected. Therefore, the accurate slice of Figure 22 **(b)** can be computed in principle. Moreover, if replacement of an entire **if** statement by one of the statements in its branches is allowed, one might even compute the minimal slice of Figure 22 **(c)**. Other compiler optimization techniques such as loop invariant motion and loop unrolling (see e.g. [87] for a comprehensive overview) may also be employed to obtain more precise slices.

Figure 23 **(a)** shows another example program, which is to be sliced with respect to its final statement `write(y)`. Once again, traditional slicing algorithms will fail to omit any statements. A more accurate slice for this example can be acquired by 'merging' the two **if** statements. The effect of this semantics-preserving transformation is shown in Figure 23 **(b)**. Clearly, a slicing algorithm which could conceptually perform this transformation would be able to determine the more accurate slice shown in Figure 23 **(c)**.

Field and Tip are currently working on a reduction-theoretical framework for computing accurate slices. Instead of performing semantics-preserving transformations on source programs, programs are first compiled into an intermediate graph representation named Pim [28]. This representation was carefully designed to accommodate transformations and

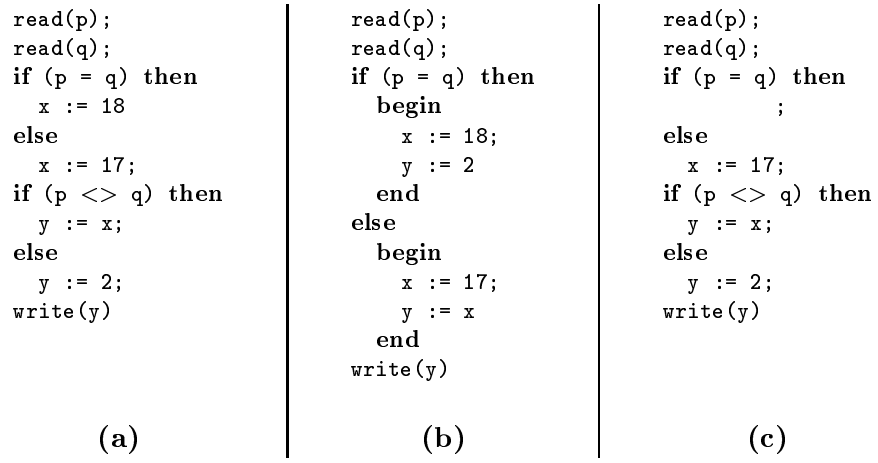


Figure 23: **(a)** Example program. **(b)** Transformed program. **(c)** More accurate slice obtained by slicing in the transformed program.

simplifications such as those shown in Figures 22 and 23. The transformation of source programs to PIM graphs, as well as subsequent optimizations and transformations on this representation are expressed by way of an *algebraic specification* [17]. Orienting the equations of this specification from left to right yields a *term rewriting system* [54]. In [29], a dynamic dependence relation for term rewriting systems is developed which can be used to keep track of 'corresponding' parts of a source program, the intermediate representation it compiles to, and the optimized version of that PIM-graph. Roughly speaking, a PIM graph contains a subgraph for each statement that represents its store expression. Slices are computed in this framework by selecting a store expression in the optimized PIM graph, and tracing the dependence relations back to the source program.

Both PIM and dynamic dependence relations have been implemented using the ASF+SDF programming environment generator [53] developed at CWI. Recent experiments have produced promising results. In particular, the (accurate) slices of Figures 22 **(b)** and 23 **(c)** have been computed.

6 Applications of Program Slicing

6.1 Debugging and Program Analysis

Debugging can be a difficult task when one is confronted with a large program, and little clues regarding the location of a bug. Program slicing is useful for debugging, because it potentially allows one to ignore many statements in the process of localizing a bug [64]. If a program computes an erroneous value for a variable x, only the statements in the slice w.r.t. x have (possibly) contributed to the computation of that value; all statements which are not in the slice can safely be ignored.

Forward slices are also useful for debugging. Suppose that, in the course of debugging, statement s is found to be incorrect. Before making a change to s, one could examine the forward slice w.r.t. s, indicating the program parts affected by s. This may produce useful insights how the error may be corrected.

Lyle and Weiser [65] introduce *program dicing*, a method for combining the information

of different slices. The basic idea is that, when a program computes a correct value for variable x and an incorrect value for variable y, the bug is *likely* to be found in statements which are in the slice w.r.t. y but not in the slice w.r.t. x. This approach is not fail-safe in the presence of multiple bugs, and when computations that use incorrect values produce correct values (referred to as *coincidental correctness* in [1]). The authors claim that program dicing still produces useful results when these assumptions are relaxed.

Static slicing methods can detect 'dead' code, i.e., statements which cannot affect any output of the program [16]. Often, such statements are not executable because of the presence of a bug. Static slicing can also be used to determine uninitialized variables which are used in expressions, another symptom of an error in the program [16].

In debugging, one is often interested in a specific execution of a program that exhibits anomalous behavior. *Dynamic* slices are particularly useful here, because they only reflect the actual dependences of that execution, resulting in smaller slices than static ones. Agrawal's thesis [1] contains a detailed discussion how static and dynamic [3, 5] slicing can be utilized for semi-automated debugging of programs. He proposes an approach where the user gradually 'zooms out' from the location where the bug manifested itself by repeatedly considering larger data and control slices. A *data slice* is obtained by only taking (static or dynamic) data dependences into account; a *control slice* consists of the set of control predicates surrounding a language construct. The closure of all data and control slices w.r.t. an expression is the (static or dynamic) slice w.r.t. the set of variables used in the expression. The information of several dynamic slices can be combined to gain some insight into the location of a bug. In [1], several operations on slices are proposed to this end, such as union, intersection, and difference. The difference operation is a dynamic version of the program 'dicing' notion of [65]. Obviously, these operations for combining slices may produce misleading information in the presence of multiple bugs or coincidental correctness. In [4], implementation of a debugging tool based on the ideas in [1, 3, 5] is discussed.

In [22], Choi, Miller and Netzer describe the design and efficient implementation of a debugger for parallel programs which incorporates *flowback analysis*, a notion introduced in the seminal paper by Balzer [10]. Intuitively, flowback analysis reveals how the computation of values depends on the earlier computation of other values. The difference between flowback analysis and (dependence graph based) dynamic slices is that the former notion allows one to interactively browse through a dependence graph, whereas the latter consists of the set of all program parts corresponding to vertices of the graph from which a designated vertex—the criterion—can be reached.

Fritzson et al. use interprocedural static [30] and dynamic [49, 52] slicing for algorithmic debugging [77, 78]. An algorithmic debugger partially automates the task of localizing a bug by comparing the *intended* program behavior with the *actual* program behavior. The intended behavior is obtained by asking the user whether or not a program unit (e.g., a procedure) behaves correctly. Using the answers given by the user, the location of the bug can be determined at the unit level. By applying the algorithmic debugging process to a *slice* w.r.t. an incorrectly valued variable instead of the entire program, many irrelevant questions can be skipped.

6.2 Program Differencing and Program Integration

Program *differencing* [37] is the task of analyzing an old and a new version of a program in order to determine the set of program components of the new version that represent syntactic and semantic changes. Such information is useful because only the program components reflecting changed behavior need to be tested. The key issue in program differencing consists of partitioning the components of the old and new version in a way that two components are in the same partition only if they have equivalent behaviors. The program integration algorithm of [44] discussed below, compares slices in order to detect equivalent behaviors. However, a partitioning technique presented in [37], which is not based on comparing slices, produces more accurate results because semantics-preserving transformations can be accommodated.

Horwitz, Prins, and Reps use the static slicing algorithm for single-procedure programs of [44] as a basis for an algorithm that integrates changes in variants of a program [41]. The inputs of their algorithm consist of a program *Base*, and two variants A and B which have been derived from *Base*. The algorithm consists of the following steps:

1. The PDGs G_{Base}, G_A, and G_B are constructed. Correspondences between 'related' vertices of these graphs are assumed to be available.

2. Sets of *affected points* of G_A and G_B w.r.t. G_{Base} are determined; these consist of vertices in G_A (G_B) which have a different slice in G_{Base}[19].

3. A merged PDG G_M is constructed from G_A, G_B, and the sets of affected points determined in (2).

4. Using G_A, G_B, G_M, and the sets of affected points computed in (2), the algorithm determines whether or not the behaviors of A and B are preserved in G_M. This is accomplished by comparing the slices w.r.t. the affected points of G_A (G_B) in G_M and G_A (G_B). If different slices are found, the changes interfere and the integration cannot be performed.

5. If the changes in A and B do not interfere, the algorithm tests if G_M is a feasible PDG, i.e., if it corresponds to some program. If this is the case, program M is constructed from G_M. Otherwise, the changes in A and B cannot be integrated.

A semantic justification for the single-procedure slicing algorithm of [44] and the program integration algorithm of [41] is presented in [76]. This paper formalizes the relationship between the execution behaviors of programs, slices of those programs, and between variants of a program and the corresponding integrated version. The comparison of slices (in step 4) relies on the existence of a mapping between the different components. If such a mapping were not available, however, the techniques of [42] for comparing two slices in linear time of the sum of their sizes could be used.

An alternative formulation of the Horwitz-Prins-Reps program integration algorithm, based on Brouwerian algebras, is presented in [71]. The algebraic laws that hold in such algebras are used to restate the algorithm and to prove properties such as associativity of consecutive integrations.

[19] These sets of affected points can be computed efficiently by way of a *forward* slice w.r.t. all *directly* affected points, i.e., all vertices in G_A that do not occur in G_{Base} and all vertices in that have a different set of incoming edges in G_A and in G_{Base} [43].

6.3 Software Maintenance

One of the problems in software maintenance consists of determining whether a change at some place in a program will affect the behavior of other parts of the program. In [31, 32], Gallagher and Lyle use static slicing for the decomposition of a program into a set of components (i.e., reduced programs), each of which captures part of the original program's behavior. They present a set of guidelines for the maintainer of a component which, if obeyed, preclude changes of the behavior of other components. Moreover, they describe how changes in a component can be merged back into the complete program in a semantically consistent way.

Gallagher and Lyle use the notion of a *decomposition slice* for the decomposition of programs. A decomposition slice w.r.t. a variable v consists of all statements that may affect the 'observable' value of v at some point; it is defined as the union of the slices w.r.t. v at any statement that outputs v, and the the last statement of the program. An *output-restricted* decomposition slice (ORD slice) is a decomposition slice from which all output statements are removed. Two ORD slices are *independent* if they have no statements in common; an ORD slice is *strongly dependent* on another ORD slice if it is a subset of the latter. An ORD slice which is not strongly dependent on any other ORD slice is *maximal*. A statement which occurs in more than one ORD slice is *dependent*; otherwise it is *independent*. A variable is *dependent* if it is assigned to in some dependent statement; it is *independent* if it is only assigned to in independent statements. Only maximal ORD slices contain independent statements, and the union of all maximal ORD slices is equal to the original program (minus output statements). The *complement* of an ORD slice is defined as the original program minus all independent statements of the ORD slice and all output statements. Intuitively, a decomposition slice captures part of the behavior of a program, and its complement captures the behavior of the rest of the program.

The essential observation of [32] is that independent statements in a slice do not affect the data and control flow in the complement. This results in the following guidelines for modification:

- Independent statements may be deleted from a decomposition slice.
- Assignments to independent variables may be added anywhere in a decomposition slice.
- Logical expressions and output statements may be added anywhere in a decomposition slice.
- New control statements that surround any dependent statements will affect the complement's behavior.

New variables may be considered as independent variables, provided that there are no name clashes with variables in the complement. If changes are required that involve a dependent variable v, the user can either extend the slice so that v is independent (in a way described in the paper), or introduce a new variable. Merging changes to components into the complete program is a trivial task. Since it is guaranteed that changes to an ORD slice do not affect its complement, only testing of the modified slice is necessary.

6.4 Testing

A program satisfies a 'conventional' *data flow testing* criterion if all def-use pairs occur in a successful test-case. Duesterwald, Gupta, and Soffa propose a more rigorous testing criterion, based on program slicing in [26]: each def-use pair must be exercised in a successful test-case; moreover it must be *output-influencing*, i.e., have an influence on at least one output value. A def-use pair is output-influencing if it occurs in an *output slice*, i.e., a slice w.r.t. an output statement. It is up to the user, or an automatic test-case generator to construct enough test-cases such that all def-use pairs are tested. Three slicing approaches are utilized, based on different dependence graphs. Static slices are computed using *static dependence graphs* (similar to the PDGs in [44]), dynamic slices are computed using *dynamic dependence graphs* (similar to [5], but instances of the same vertex are merged, resulting in a slight loss of precision), and *hybrid slices* are computed using dependence graphs with combined static and dynamic information (similar to the quasi-static slices in [79]). In the hybrid approach, the set of variables in the program is partitioned into two disjoint subsets in a way that variables in one subset do not refer to variables in the other subset. Static dependences are computed for one subset (typically scalar variables), dynamic dependences for the other subset (typically arrays and pointers). The advantage of this approach is that it combines reasonable efficiency with reasonable precision.

In [50], Kamkar, Shahmehri, and Fritzson extend the work of Duesterwald, Gupta, and Soffa to multi-procedure programs. To this end, they define appropriate notions of interprocedural def-use pairs. The interprocedural dynamic slicing method of [51, 52] is used to determine which interprocedural def-use pairs have an effect on a correct output value, for a given test case. The summary graph representation of [51, 52] (see Section 4.2) is slightly modified by annotating vertices and edges with def-use information. This way, the set of def-use pairs exercised by a slice can be determined efficiently.

Regression testing consists of re-testing only the parts affected by a modification of a previously tested program, while maintaining the 'coverage' of the original test suite. Gupta, Harrold, and Soffa describe an approach to regression testing where slicing techniques are used [34]. Backward and forward static slices serve to determine the program parts affected by the change, and only test cases which execute 'affected' def-use pairs need to be executed again. Conceptually, slices are computed by backward and forward traversals of the CFG of a program, starting at the point of modification. However, the algorithms in [34] are designed to determine the information necessary for regression testing only (i.e., affected def-use pairs).

In [14], Bates and Horwitz use a variation of the PDG notion of [41] for incremental program testing. Testing criteria are defined in terms of PDG notions: i.e., the all-vertices testing criterion is satisfied if each vertex of the PDG is exercised by a test set (i.e., each statement and control predicate in the program is executed). An all-flow-edges criterion is defined in a similar manner. Given a tested and subsequently modified program, slicing is used to determine: (i) the statements affected by the modification, and (ii) the test-cases that can be reused for the modified program. Roughly speaking, the former consists of the statements which did not occur previously as well as and the statements which have different slices. The latter requires partitioning the statements of the original and the modified program into equivalence classes; statements are in the same class if they have the same 'control' slice (a slightly modified version of the standard notion). Bates and Horwitz proof that statements in the same class are exercised by the same test cases.

6.5 Tuning Compilers

Larus and Chandra present an approach to the tuning of compilers where dynamic slicing is used to detect potential occurrences of redundant common subexpressions [62]. Finding such a common subexpression is an indication of sub-optimal code being generated.

Object code is instrumented with trace-generating instructions. A trace-regenerator reads a trace and produces a stream of events, such as the read and load of a memory location. This stream of events is input for a compiler-auditor (e.g., a common-subexpression elimination auditor) which constructs dynamic slices w.r.t. the current values stored in registers. Larus and Chandra use a variant of the approach in [5]: a dynamic slice is represented by directed acyclic graph (DAG) containing all operators and operands that produced the current value in a register. A common subexpression occurs when isomorphic DAGs are constructed for two registers. However, the above situation only indicates that a common subexpression occurs in a *specific* execution. A common subexpression occurs in *all* execution paths if its inputs are the same in all executions. This is verified by checking that: (i) the program counter PC1 for the first occurrence of the common subexpression dominates the program counter PC2 for the second occurrence, (ii) the register containing the first occurrence of the common subexpression is not modified along any path between PC1 and PC2, and (iii) neither are the inputs to the common subexpression modified along any path between PC1 and PC2. Although the third condition is impossible to verify in general, it is feasible to do so for a number of special cases. In general, it is up to the compiler writer to check condition (iii).

6.6 Other Applications

Weiser describes how slicing can be used to *parallelize* the execution of a sequential program [84]. Several slices of a program are executed in parallel, and the outputs of the slices are *spliced* together in such a way that the I/O behavior of the original program is preserved. In principle, the splicing process may take place in parallel with the execution of the slices. A natural requirement of Weiser's splicing algorithm is that the set of all slices should 'cover' the execution behavior of the original program. Splicing does not rely on a particular slicing technique; any method which computes executable static slices is adequate. Only programs with structured control flow are considered, because Weiser's splicing algorithm depends on the fact that execution behavior can be expressed in terms of a so-called program regular expression. The main reason for this is that reconstruction of the original I/O behavior becomes unsolvable in the presence of irreducible control flow.

Ott and Thus view a module as a set of processing elements which act together to compute the outputs of a module. They classify the *cohesion class* of a module (i.e, the kind of relationships between the processing elements) by comparing the slices w.r.t. different output variables [68]. *Low* cohesion corresponds to situations where a module is partitioned into disjoint sets of unrelated processing elements. Each set is involved in the computation of a different output value, and there is no overlap between the slices. *Control* cohesion consists of two or more sets of disjoint processing elements each of which depends on a common input value; the intersection of slices will consist of control predicates. *Data* cohesion corresponds to situations where data flows from one set of processing elements to another; slices will have non-empty intersection and non-trivial differences. *High* cohesion situations resemble pipelines. The data from a processing element flows to its successor;

the slices of high cohesion modules will overlap to a very large extent. The paper does not rely on any specific slicing method, and no quantitative measures are presented.

In [18], Binkley presents a graph rewriting semantics for System Dependence Graphs which he uses to perform interprocedural constant propagation. The interprocedural slicing algorithm of [44] is used to extract slices that may be executed to obtain constant values.

In [15], Beck and Eichmann consider the case where a 'standard' module for an abstract data type module is used, and where only part of its functionality is required. Their objective is to 'slice away' all unnecessary code in the module. To this end, they generalize the notion of static slicing to modular programs. In order to compute a reduced version of a module, an *interface dependence graph* (IDG) is constructed. This graph contains vertices for all definitions of types and global variables, and subprograms inside a module. Moreover, the IDG contains edges for every def-use relation between vertices. An *interface slicing criterion* consists of a module and a subset of the operations of the ADT. Computing interface slices corresponds to solving a reachability problem in an IDG. Inter-module slices, corresponding to situations where modules import other modules, can be computed by deriving new criteria for the imported modules.

Ning, Engberts, and Kozaczynski discuss a set of tools for extracting components from large Cobol systems. These tools include facilities for *program segmentation*, i.e., distinguishing pieces of functionally related code. In addition to backward and forward static slices, *condition-based* slices can be determined. For a condition-based slice, the criterion specifies a constraint on the values of certain variables.

7 Conclusions

We have presented a survey of the static and dynamic slicing techniques that can be found in the present literature. As a basis for classifying slicing techniques we have used the computation method, and a variety of programming language features such as procedures, arbitrary control flow, composite variables/pointers, and interprocess communication. Essentially, the problem of slicing in the presence of one of these features is 'orthogonal' to solutions for each of the other features. For dynamic slicing methods, an additional issue is the fact whether or not the computed slices are *executable* programs which capture a part of the program's behavior. Wherever possible, we have compared different solutions to the same problem by applying each algorithm to the same example program. In addition we have discussed the possibilities and problems associated with the integration of solutions for 'orthogonal' language features.

In Section 3.6, we have compared and classified algorithms for static slicing. Besides listing the specific slicing problems studied in the literature, we have compared the *accuracy* and *efficiency* of static slicing algorithms. The most significant conclusions of Section 3.6 can be summarized as follows:

basic algorithms For *intra*procedural static slicing in the absence of procedures, unstructured control flow, composite datatypes and pointers, and interprocess communication, the accuracy of methods based on dataflow equations [85], information-flow relations [16], and program dependence graphs [69] is essentially the same. PDG-based algorithms have the advantage that dataflow analysis has to be performed

only once; after that, slices can be extracted in linear time. This is especially useful when several slices of the same program are required.

procedures The first solution for *inter*procedural static slicing, presented by Weiser in [85], is inaccurate for two reasons. First, this algorithm does not use exact dependence relations between input and output parameters. Second, the call-return structure of execution paths is not taken into account. We have shown in Section 3.2.1 that Weiser's algorithm may slice a procedure several times in the presence of loops. The solution by Bergeretti and Carré [16] does not compute truly interprocedural slices because no procedures other than the main program are sliced. Moreover, the approach by Bergeretti and Carré is not sufficiently general to handle recursion. Exact solutions to the interprocedural static slicing problem have been presented by Hwang, Du, and Chou [45], Reps, Horwitz and Binkley [44], and Reps, Horwitz, Sagiv, and Rosay [74, 75]. The Reps-Horwitz-Sagiv-Rosay algorithm for interprocedural static slicing is the most efficient one.

arbitrary control flow Lyle was the first to present an algorithm for static slicing in the presence of arbitrary control flow [64]. The solution he presents in very conservative: it may include more **goto** statements than necessary. Agrawal has shown in [2] that solutions proposed by Gallagher and Lyle [31, 32] and by Jiang et al. are incorrect. Precise solutions for static slicing in the presence of arbitrary control flow have been proposed by Ball and Horwitz [8, 9], Choi and Ferrante [21], and Agrawal [2]. It is not clear how the efficiency of these algorithms compares.

composite datatypes/pointers Lyle has presented a conservative algorithm for static slicing in the presence of arrays in [64]. The algorithm proposed by Jiang et al. in [47] is incorrect. Agrawal, DeMillo, and Spafford propose a PDG-based algorithm for static slicing in the presence of composite variables and pointers.

interprocess communication The only approach for static slicing of concurrent programs was proposed by Cheng [19]. Unfortunately, Cheng has not provided a justification of the correctness of his algorithm.

We have compared and classified algorithms for dynamic slicing in Section 4.5. Due to differences in computation methods and dependence graph representations, the potential for integration of the dynamic slicing solutions for 'orthogonal' problems is less clear than in the static case. The conclusions of Section 4.5 may be summarized as follows:

basic algorithms Methods for *intra*procedural dynamic slicing in the absence of procedures, composite datatypes and pointers, and interprocess communication were proposed by Korel and Laski [56, 57], Agrawal and Horgan [5], and Gopal [33]. The slices determined by the Agrawal-Horgan algorithm and the Gopal algorithm are more accurate than the slices computed by the Korel-Laski algorithm, because Korel and Laski insist that their slices be executable programs. The Korel-Laski algorithm and Gopal's algorithm require an unbounded amount of space because the entire execution history of the program has to be stored. Since slices are computed by traversing this history, the amount of time needed to compute a slice depends on the number of executed statements. A similar statement can be made for the flowback analysis algorithm of [22, 67]. The algorithm proposed by Agrawal and

Horgan based on Reduced Dynamic Dependence Graphs requires at most $O(2^n)$ space, where n is the number of statements in the program. However, the time needed by the Agrawal-Horgan algorithm also depends on the number of executed statements because for each executed statement, the dependence graph may have to be updated.

procedures Two dependence graph based algorithms for interprocedural dynamic slicing were proposed by Agrawal, DeMillo, and Spafford [3], and by Kamkar, Shahmehri, and Fritzson [51, 52]. The former method relies heavily on the use of memory cells as a basis for computing dynamic reaching definitions. Various procedure-passing mechanisms can be modeled easily by assignments of actual to formal and formal to actual parameters at the appropriate moments. The latter method is also expressed as a reachability problem in a (summary) graph. However, there are a number of differences with the approach of [3]. First, parts of the graph can be constructed at compile-time. This is more efficient, especially in cases where many calls to the same procedure occur. Second, Kamkar et al. study procedure-level slices; that is, slices consisting of a set of procedure calls rather than a set of statements. Third, the size of a summary graph depends on the number of executed procedure calls, whereas the graphs of [3] are more space efficient due to 'fusion' of vertices with the same transitive dependences. It is unclear if one algorithm produces more precise slices than the other.

arbitrary control flow As far as we know, dynamic slicing in the presence of arbitrary control flow has not been studied yet. However, we conjecture that the solutions for the static case [2, 8, 9, 21] may be adapted for dynamic slicing.

composite datatypes/pointers Two approaches for dynamic slicing in the presence of composite datatypes and pointers were proposed, by Korel and Laski [57], and Agrawal, DeMillo, and Spafford [3]. The algorithms differ in their computation method: dynamic flow concepts vs. dependence graphs, and in the way composite datatypes and pointers are represented. Korel and Laski treat components of composite datatypes as distinct variables, and invent names for dynamically allocated objects and pointers whereas Agrawal, DeMillo, and Spafford base their definitions on definitions and uses of memory cells. It is unclear how the accuracy of these algorithms compares. The time and space requirements of both algorithms are essentially the same as in the case where only scalar variables occur.

interprocess communication Several methods for dynamic slicing of distributed programs have been proposed. Korel and Ferguson [55] and Duesterwald, Gupta, and Soffa [25] compute slices that are executable programs, but have a different way of dealing with nondeterminism in distributed programs: the former approach requires a mechanism for replaying the rendezvous in the slice in the same relative order as they occurred in the original program whereas the latter approach replaces nondeterministic communication statements in the program by deterministic communication statements in the slice. Cheng [19] and Choi et al. [22, 67] do not consider this problem because the slices they compute are not executable programs. Duesterwald, Gupta, and Soffa [25] and Cheng [19] use *static* dependence graphs for computing *dynamic* slices. Although this is more space-efficient than the other ap-

proaches, the computed slices will be inaccurate (see the discussion in Section 4.1.1). The algorithms by Korel and Ferguson and by Choi et al. both require an amount of space that depends on the number of executed statements. Korel and Ferguson require their slices to be executable; therefore these slices will contain more statements than those computed by the algorithm of [22, 67].

In Section 5, we have argued that compiler-optimization techniques and semantics-preserving program transformations can be used to obtain more accurate slices [29].

In Section 6, we have presented an overview how slicing techniques are applied in the areas of debugging, program analysis, program integration, software maintenance, dataflow testing, and others.

Acknowledgements

I am grateful to John Field, Jan Heering, Susan Horwitz, Paul Klint, G. Ramalingam, and Tom Reps for many fruitful discussions and comments on earlier drafts of this paper. Tom Reps provided the program and picture of Figure 10. Susan Horwitz provided the program of Figure 13. The programs shown in Figures 2 and 16 are adaptations of example programs in [1].

References

[1] Agrawal, H. *Towards automatic debugging of Computer Programs.* PhD thesis, Purdue University, 1992.

[2] Agrawal, H. On slicing programs with jump statements. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation* (Orlando, Florida, 1994). To appear.

[3] Agrawal, H., DeMillo, R., and Spafford, E. Dynamic slicing in the presence of unconstrained pointers. In *Proceedings of the ACM Fourth Symposium on Testing, Analysis, and Verification (TAV4)* (1991), pp. 60–73. Also Purdue University technical report SERC-TR-93-P.

[4] Agrawal, H., DeMillo, R., and Spafford, E. Debugging with dynamic slicing and backtracking. *Software—Practice and Experience 23*, 6 (1993), 589–616.

[5] Agrawal, H., and Horgan, J. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation* (1990), pp. 246–256. *SIGPLAN Notices* 25(6).

[6] Aho, A., Sethi, R., and Ullman, J. *Compilers. Principles, Techniques and Tools.* Addison-Wesley, 1986.

[7] Alpern, B., Wegman, M., and Zadeck, F. Detecting equality of variables in programs. In *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages* (San Diego, 1988), pp. 1–11.

[8] Ball, T. *The Use of Control-Flow and Control Dependence in Software Tools.* PhD thesis, University of Wisconsin-Madison, 1993.

[9] Ball, T., and Horwitz, S. Slicing programs with arbitrary control-flow. In *Proceedings of the First International Workshop on Automated and Algorithmic Debugging* (1993), P. Fritzson, Ed., vol. 749 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 206–222.

[10] Balzer, R. EXDAMS - EXtendable Debugging And Monitoring System. In *Proceedings of the AFIPS SJCC* (1969), vol. 34, pp. 567–586.

[11] Banning, J. 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* (1979), pp. 29–41.

[12] Barnes, J. *Programming in Ada*, second ed. International Computer Science Series. Addison-Wesley, 1982.

[13] Barth, J. A practical interprocedural data flow analysis algorithm. *Communications of the ACM 21*, 9 (1978), 724–736.

[14] Bates, S., and Horwitz, S. Incremental program testing using program dependence graphs. In *Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages* (Charleston, SC, 1993), pp. 384–396.

[15] Beck, J., and Eichmann, D. Program and interface slicing for reverse engineering. In *Proceedings of the 15th International Conference on Software Engineering* (Baltimore, 1993).

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

[17] Bergstra, J., Heering, J., and Klint, P., Eds. *Algebraic Specification.* ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989.

[18] Binkley, D. Interprocedural constant propagation using dependence graphs and a data-flow model. In *Proceedings of the 5th International Conference on Compiler Construction—CC'94* (Edinburgh, UK, 1994), P. Fritzson, Ed., vol. 786 of *LNCS*, pp. 374–388.

[19] Cheng, J. Slicing concurrent programs – a graph-theoretical approach. In *Proceedings of the First International Workshop on Automated and Algorithmic Debugging* (1993), P. Fritzson, Ed., vol. 749 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 223–240.

[20] Choi, J.-D., Burke, M., and Carini, P. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages* (1993), ACM, pp. 232–245.

[21] Choi, J.-D., and Ferrante, J. Static slicing in the presence of GOTO statements. *ACM Transactions on Programming Languages and Systems* (May 1994). To Appear.

[22] Choi, J.-D., Miller, B., and Netzer, R. Techniques for debugging parallel programs with flowback analysis. *ACM Transactions on Programming Languages and Systems 13*, 4 (1991), 491–530.

[23] Cooper, K., and Kennedy, K. Interprocedural side-effect analysis in linear time. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, 1988), pp. 57–66. *SIGPLAN Notices* 23(7).

[24] Cytron, R., Ferrante, J., and Sarkar, V. Compact representations for control dependence. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation* (White Plains, New York, 1990), pp. 337–351. *SIGPLAN Notices* 25(6).

[25] Duesterwald, E., Gupta, R., and Soffa, M. Distributed slicing and partial re-execution for distributed programs. In *Proceedings of the fifth workshop on Languages and Compilers for Parallel Computing* (New Haven, Connecticut, 1992), pp. 329–337.

[26] Duesterwald, E., Gupta, R., and Soffa, M. Rigorous data flow testing through output influences. In *Proceedings of the Second Irvine Software Symposium ISS'92* (California, 1992), pp. 131–145.

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

[28] Field, J. A simple rewriting semantics for realistic imperative programs and its application to program analysis. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation* (1992), pp. 98–107.

[29] Field, J., and Tip, F. Dynamic dependence in term rewriting systems and its application to program slicing. Report CS-R94xx, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1994. Forthcoming. Also to appear in proceedings of PLILP '94.

[30] Fritzson, P., Shahmehri, N., Kamkar, M., and Gyimothy, T. Generalized algorithmic debugging and testing. *ACM Letters on Programming Languages and Systems 1*, 4 (1992), 303–322.

[31] Gallagher, K. *Using Program Slicing in Software Maintenance.* PhD thesis, University of Maryland, 1989.

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

[33] Gopal, R. Dynamic program slicing based on dependence relations. In *Proceedings of the Conference on Software Maintenance* (1991), pp. 191–200.

[34] Gupta, R., Harrold, M., and Soffa, M. An approach to regression testing using slicing. In *Proceedings of the Conference on Software Maintenance* (1992), pp. 299–308.

[35] Gupta, R., and Soffa, M. A framework for generalized slicing. Technical report TR-92-07, University of Pittsburgh, 1992.

[36] Hausler, P. Denotational program slicing. In *Proceedings of the 22nd Hawaii International Conference on System Sciences* (Hawaii, 1989), pp. 486–494.

[37] Horwitz, S. Identifying the semantic and textual differences between two versions of a program. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation* (White Plains, New York, 1990), pp. 234–245. *SIGPLAN Notices* 25(6).

[38] Horwitz, S., Pfeiffer, P., and Reps, T. Dependence analysis for pointer variables. In *Proceedings of the ACM 1989 Conference on Programming Language Design and Implementation* (Portland, Oregon, 1989). *SIGPLAN Notices* 24(7).

[39] Horwitz, S., Prins, J., and Reps, T. Integrating non-interfering versions of programs. In *Conference Record of the ACM SIGSOFT/SIGPLAN Symposium on Principles of Programming Languages* (1988), pp. 133–145.

[40] Horwitz, S., Prins, J., and Reps, T. On the adequacy of program dependence graphs for representing programs. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages* (1988), ACM, pp. 146–157.

[41] Horwitz, S., Prins, J., and Reps, T. Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems 11*, 3 (1989), 345–387.

[42] Horwitz, S., and Reps, T. Efficient comparison of program slices. *Acta Informatica 28* (1991), 713–732.

[43] Horwitz, S., and Reps, T. The use of program dependence graphs in software engineering. In *Proceedings of the 14th International Conference on Software Engineering* (Melbourne, Australia, 1992), pp. 392–411.

[44] Horwitz, S., Reps, T., and Binkley, D. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems 12*, 1 (1990), 26–61.

[45] Hwang, J., Du, M., and Chou, C. Finding program slices for recursive procedures. In *Proceedings of the 12th Annual International Computer Software and Applications Conference* (Chicago, 1988).

[46] Hwang, J., Du, M., and Chou, C. The influence of language semantics on program slices. In *Proceedings of the 1988 International Conference on Computer Languages* (Miami Beach, 1988).

[47] Jiang, J., Zhou, X., and Robson, D. Program slicing for C - the problems in implementation. In *Proceedings of the Conference on Software Maintenance* (1991), pp. 182–190.

[48] Kamkar, M. An overview and comparative classification of static and dynamic program slicing. Technical Report LiTH-IDA-R-91-19, Linköping University, Linköping, 1991. To appear in *Journal of Systems and Software*.

[49] Kamkar, M. *Interprocedural Dynamic Slicing with Applications to Debugging and Testing*. PhD thesis, Linköping University, 1993.

[50] Kamkar, M., Fritzson, P., and Shahmehri, N. Interprocedural dynamic slicing applied to interprocedural data flow testing. In *Proceedings of the Conference on Software Maintenance* (Montreal, Canada, 1993), pp. 386–395.

[51] Kamkar, M., Fritzson, P., and Shahmehri, N. Three approaches to interprocedural dynamic slicing. *Microprocessing and Microprogramming 38* (1993), 625–636.

[52] Kamkar, M., Shahmehri, N., and Fritzson, P. Interprocedural dynamic slicing. In *Proceedings of the 4th International Conference on Programming Language Implementation and Logic Programming* (1992), M. Bruynooghe and M. Wirsing, Eds., vol. 631 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 370–384.

[53] Klint, P. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology 2*, 2 (1993), 176–201.

[54] Klop, J. Term rewriting systems. In *Handbook of Logic in Computer Science, Volume 2. Background: Computational Structures*, S. Abramsky, D. Gabbay, and T. Maibaum, Eds. Oxford University Press, 1992, pp. 1–116.

[55] Korel, B., and Ferguson, R. Dynamic slicing of distributed programs. *Applied Mathematics and Computer Science 2*, 2 (1992), 199–215.

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

[57] Korel, B., and Laski, J. Dynamic slicing of computer programs. *Journal of Systems and Software 13* (1990), 187–195.

[58] Kuck, D., Kuhn, R., Padua, D., Leasure, B., and Wolfe, M. Dependence graphs and compiler optimizations. In *Conference Record of the Eighth ACM Symposium on Principles of Programming Languages* (1981), pp. 207–218.

[59] Lakhotia, A. Graph theoretic foundations of program slicing and integration. Report CACS TR-91-5-5, University of Southwestern Louisiana, 1991.

[60] Lakhotia, A. Improved interprocedural slicing algorithm. Report CACS TR-92-5-8, University of Southwestern Louisiana, 1992.

[61] Landi, W., and Ryder, B. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the 1992 ACM Conference on Programming Language Design and Implementation* (San Francisco, 1992), pp. 235–248. *SIGPLAN Notices* 27(7).

[62] Larus, J., and Chandra, S. Using tracing and dynamic slicing to tune compilers. Computer sciences technical report #1174, University of Wisconsin-Madison, 1993.

[63] Leung, H., and Reghbati, H. Comments on program slicing. *IEEE Transactions on Software Engineering SE-13*, 12 (1987), 1370–1371.

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

[65] Lyle, J., and Weiser, M. Automatic bug location by program slicing. In *Proceedings of the Second International Conference on Computers and Applications* (Beijing (Peking), China, 1987), pp. 877–883.

[66] Maydan, D., Hennessy, J., and Lam, M. Efficient and exact data dependence analysis. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation* (1991), pp. 1–14. *SIGPLAN Notices* 26(6).

[67] Miller, B., and Choi, J.-D. A mechanism for efficient debugging of parallel programs. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation* (Atlanta, 1988), pp. 135–144. *SIGPLAN Notices* 23(7).

[68] Ott, L. M., and Thuss, J. The relationship between slices and module cohesion. In *Proceedings of the 11th International Conference on Software Engineering* (1989), pp. 198–204.

[69] Ottenstein, K., and Ottenstein, L. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (1984), pp. 177–184. *SIGPLAN Notices* 19(5).

[70] Podgurski, A., and Clarke, L. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering 16*, 9 (1990), 965–979.

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

[72] Reps, T. On the sequential nature of interprocedural program-analysis problems. Unpublished report, University of Copenhagen, 1994.

[73] Reps, T., and Bricker, T. Illustrating interference in interfering versions of programs. In *Proceedings of the Second International Workshop on Software Configuration Management* (Princeton, 1989), pp. 46–55. *ACM SIGSOFT Software Engineering Notes* Vol.17 No.7.

[74] Reps, T., Horwitz, S., Sagiv, M., and Rosay, G. Speeding up slicing. Unpublished report, Datalogisk Institut, University of Copenhagen, 1994.

[75] Reps, T., Sagiv, M., and Horwitz, S. Interprocedural dataflow analysis via graph reachability. Report DIKU TR 94-14, University of Copenhagen, Copenhagen, 1994.

[76] Reps, T., and Yang, W. The semantics of program slicing and program integration. In *Proceedings of the Colloquium on Current Issues in Programming Languages* (1989), vol. 352 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 60–74.

[77] Shahmehri, N. *Generalized Algorithmic Debugging*. PhD thesis, Linköping University, 1991.

[78] Shapiro, E. *Algorithmic Program Debugging*. MIT Press, 1982.

[79] Venkatesh, G. The semantic approach to program slicing. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation* (1991), pp. 107–119. *SIGPLAN Notices* 26(6).

[80] Wegman, M., and Zadeck, F. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems 13*, 2 (1991), 181–210.

[81] Weihl, W. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *Conference Record of the Seventh ACM Symposium on Principles of Programming Languages* (1980), pp. 83–94.

[82] Weiser, M. *Program slices: formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, 1979.

[83] Weiser, M. Programmers use slices when debugging. *Communications of the ACM 25*, 7 (1982), 446–452.

[84] Weiser, M. Reconstructing sequential behavior from parallel behavior projections. *Information Processing Letters 17*, 3 (1983), 129–135.

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

[86] Weiser, M. Private communication, 1994.

[87] Zima, H., and Chapman, B. *Supercompilers for Parallel and Vector Computers*. ACM Press Frontier Series. ACM Press, New York, 1991.