*Journal of Programming Languages* 3 (1995) 121–189

# *A survey of program slicing techniques*

FRANK TIP*

*IBM T. J. Watson Research Center, PO Box 704, Yorktown Heights, NY 10598, USA*

A *program slice* consists of the parts of a program that (potentially) affect the values computed at some point of interest. Such a point of interest is referred to as a *slicing criterion*, and is typically specified by a location in the program in combination with a subset of the program's variables. 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. Static slices are computed without making assumptions regarding a program's input, whereas the computation of dynamic slices relies on a specific test case. This survey presents an overview of program slicing, including the various general approaches used to compute slices, as well as the specific techniques used to address a variety of language features such as procedures, unstructured control flow, composite data types and pointers, and concurrency. Static and dynamic slicing methods for each of these features are compared and classified in terms of their accuracy and efficiency. Moreover, the possibilities for combining solutions for different features are investigated. Recent work on the use of compiler-optimization and symbolic execution techniques for obtaining more accurate slices is discussed. The paper concludes with an overview of the applications of program slicing, which include debugging, program integration, dataflow testing, and software maintenance.

**Keywords:** program slicing; static slicing; dynamic slicing; program analysis; debugging; data dependence; control dependence; program dependence graph

## 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. Such a point of interest is referred to as a *slicing criterion*, and is typically specified by a pair (program point, set of variables). The parts of a program that have a direct or indirect effect on the values computed at a slicing criterion *C* constitute 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 [1–3]. Weiser claims that a slice corresponds to the mental abstractions that people make when they are debugging a program, and advocates the integration of program slicers in debugging environments. Various slightly different notions of program slices have since 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

*This work was carried out while the author was working at the Centrum voor Wiskunde en Informatica (CWI) in Amsterdam, The Netherlands.

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

**(a)**

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

product := 1;
while i <= n do
begin

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

write(product)
```

**(b)**

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

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 behaviour of $P$. Another common definition of a slice is a *subset* of the statements and control predicates of the program that directly or indirectly affect the values computed at the criterion, but that do not necessarily constitute an executable program. An important distinction is that between a *static* and a *dynamic* slice. The former 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, these notions are introduced in some detail.

Features of programming languages such as procedures, unstructured control flow, composite data types and pointers, and concurrency each require specific extensions of slicing algorithms. Static and dynamic slicing methods for each of these features are classified and compared in terms of accuracy and efficiency. In addition, possibilities for integrating solutions for different language features are investigated. Throughout this paper, slicing algorithms are compared by applying them to similar examples.

## 1.1 *Static slicing*

Figure 1 (a) shows an example program that 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 not relevant to the (final value of) variable `product` have been 'sliced away'.

In Weiser's approach, slices are computed by computing consecutive sets of transitively 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 [4], who restate the problem of static slicing in terms of a reachability problem in a *program dependence graph* (PDG) [5, 6]. 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

```
(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)**

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

**(b)**

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

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 discussed below utilize modified and extended versions of PDGs as their underlying program representation. Yet another approach was proposed by Bergeretti and Carré [7], 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's control flow graph (CFG) or PDG, starting at the slicing criterion. Therefore, these slices are referred to as *backward* (static) slices. Bergeretti and Carré [7] were the first to define the notion of a *forward* static slice, although Reps and Bricker [8] were the first to use this terminology. 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 [9], 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 in 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 *s* to the vertex for *t*. Recently, Choi *et al.* [11, 12] have made an efficient implementation of flowback analysis for parallel programs.

In the case of dynamic program slicing, only the dependences that occur in a *specific* execution

[1] Unless stated otherwise, 'slice' will denote 'backward slice' in this paper.

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). In other words, the difference between static and dynamic slicing is that dynamic slicing assumes *fixed* input for a program, whereas static slicing does not make assumptions regarding the input. A number of hybrid approaches, where a combination of static and dynamic information is used to compute slices, can be found in the literature. Choi *et al.* [12], Duesterwald *et al.* [13], and Kamkar [14] use static information in order to decrease the amount of computations that have to be performed at run-time. Venkatesh [15], Ning *et al.* [16], Field and Tip [17] and Field *et al.* [18] consider situations where only a *subset* of the inputs to the program are constrained.

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 Fig. 2 (a) w.r.t. criterion (8, x) consists of the entire program.

## 1.3 *Applications of slicing*

The main application that Weiser had in mind for slicing was debugging [1–3]: if a program computes an erroneous value for some variable *x* at some program point, the bug is likely to be found in the slice with respect to *x* at that point. The use of slicing for debugging was further explored by Lyle and Weiser [19], Choi *et al.* [12], Agrawal *et al.* [20], Fritzson *et al.* [21], Pan [22] and Pan and Spafford [23].

A number of other applications has since been proposed: parallelization [24], program differencing and integration [25, 26], software maintenance [27], testing [13, 28–30], reverse engineering [31– 33], and compiler tuning [34]. Section 5 contains an overview of how slicing is used in each of these application areas.

## 1.4 *Related work*

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

Venkatesh [15] 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 versus dynamic, backward versus forward, and closure versus executable. Some of the slicing methods in the literature are classified according to these criteria [3, 4, 26, 35–37].

Lakhotia [38] restates a number of static slicing methods [3, 4, 26] as well as the program integration algorithm of Horwitz *et al.* [26] in terms of operations on directed graphs. He presents a

[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 6.

uniform framework of *graph slicing*, and distinguishes between *syntactic* properties of slices that 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 some dynamic slicing methods [35, 37] may be modelled in a similar way.

Gupta and Soffa present a generic algorithm for *static slicing* and the solution of related data-flow problems (such as determining reaching definitions) that is based on performing a traversal of the control flow graph [39]. 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, or dependences that 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.

Horwitz and Reps [40] 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 PDGs [41, 26, 42, 25, 36, 43]. In addition to presenting an overview of the most significant definitions, algorithms, theorems, and complexity results, the motivation for this research is discussed in considerable detail.

An earlier classification of static and dynamic slicing methods was presented by Kamkar [44, 14]. The differences between Kamkar's work and ours may be summarized as follows. First, this work is more up-to-date and complete; for instance, Kamkar does not address any of the papers that discuss slicing in the presence of unstructured control flow [45–48] or methods for computing slices that are based on information-flow relations [7, 49]. Second, the organization of our work and Kamkar's is different. Whereas Kamkar discusses each slicing method and its applications separately, this survey is organized in terms of a number of 'orthogonal' dimensions, such as the problems posed by procedures, or composite variables, aliasing, and pointers. This approach enables us to consider combinations of solutions to different dimensions. Third, unlike Kamkar we compare the accuracy and efficiency of slicing methods (by applying them to the same or similar example programs), and attempt to determine their fundamental strengths and weaknesses (i.e., irrespective of the original presentation). Finally, Kamkar does not discuss any of the recent papers [17, 18, 50] on improving the accuracy of slicing by employing compiler-optimization techniques.

## 1.5 *Organization of this paper*

The remainder of this paper is organized as follows. Section 2 introduces 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 when needed. Section 3 contains an overview of static slicing methods. First, we consider the simple case of slicing structured programs with only scalar variables. Then, algorithms for slicing in the presence of procedures, unstructured control flow, composite variables and pointers, and concurrency are considered. Section 3.6 compares and classifies methods for static slicing. Section 4 addresses dynamic slicing methods; its organization is similar to that of Section 3. Applications of program slicing are discussed in Section 5. Section 6

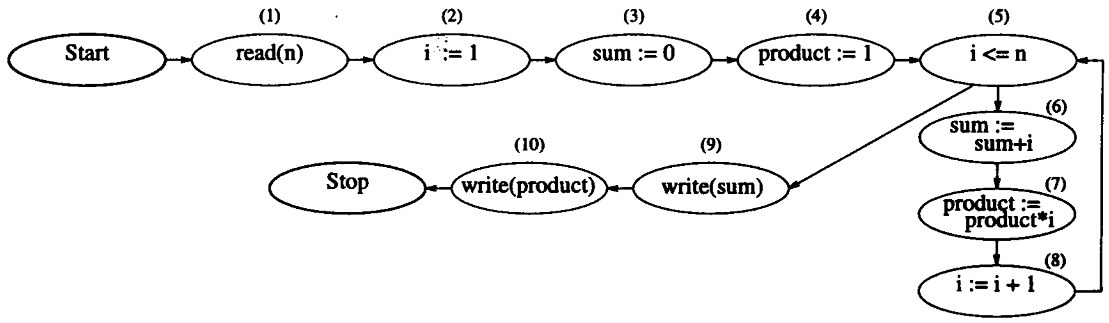


**Fig. 3.** CFG of the example program of Figure 1(a)

discusses recent work on the use of compiler-optimization techniques for obtaining more accurate slices. Finally, Section 7 summarizes the main conclusions of this survey.

## 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 labelled 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 [6]. 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. Intuitively, a statement $j$ is *flow dependent* on statement $i$ if a value computed at $i$ is used at $j$ in some program execution. In the absence of aliasing [51, 52], flow dependence may be defined formally as follows: there exists a variable $x$ such that: (i) $x \in$ DEF($i$), (ii) $x \in$ REF($j$), and, (iii) 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 $j$ if all paths from $i$ to STOP pass through $j$. A node $j$ is *control dependent* on a node $i$ if (i) there exists a path $P$ from $i$ to $j$ such that $j$ post-dominates every node in $P$, excluding $i$ and $j$, and (ii) $i$ is not post-dominated by $j$. Determining control dependences in a program with arbitrary control flow is studied by Ferrante *et al.* [6]. For programs with structured control flow, control dependences can be determined in a simple syntax-directed manner [53]: the statements immediately[3] inside the branches of an **if** or **while** statement are control dependent on the control predicate.

[3] A statement in a branch of an **if** statement that occurs within another **if** statement is only control dependent on the predicate of the inner **if** statement.

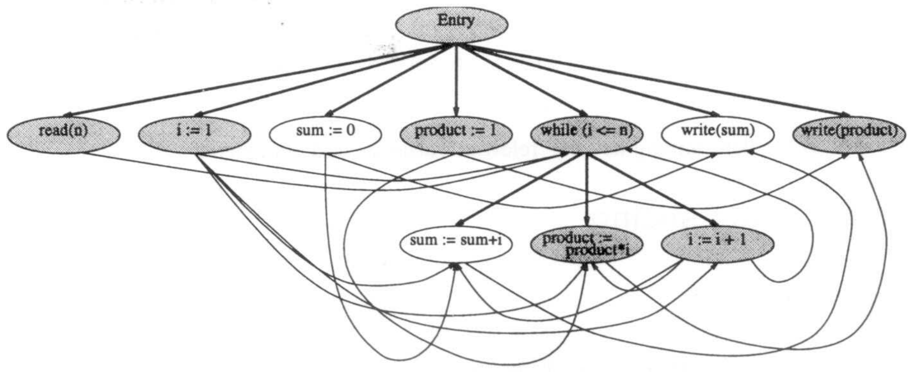


**Fig. 4.** PDG of the program in Fig. 1 (a)

As an example, Fig. 3 shows the CFG for the example program of Fig. 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.

Many of the slicing approaches that will be discussed in the sequel use the Program Dependence Graph (PDG) representation of a program [5, 6]. The vertices of the PDG correspond to the statements and control predicates of the program, and the edges of a PDG correspond to data and control dependences between them. The dependence edges of a PDG define a partial ordering on the statements of the program; the statements must be executed in this order to preserve the semantics of the program.

In the PDGs of Horwitz *et al.* [53, 26, 43, 36], 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. 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 [53] is minimal in the sense that removing any of the types of dependence edges, or disregarding the distinction between loop-carried and loop-independent flow edges would result in inequivalent programs having isomorphic PDGs. However, for the computation of program slices, only flow dependences and control dependences are necessary. Therefore, only these dependences will be considered in the sequel.

As an example, Fig. 4 shows the PDG of the program of Fig. 1 (a). In this figure, the PDG variant of Horwitz *et al.* [36] is used. Thick edges represent control dependences[4] and thin edges represent flow dependences. The shading of certain vertices in the PDG of Fig. 4 will be explained in Section 3.1.3.

[4] The usual labelling of control dependence edges is omitted here, as this is irrelevant for the present discussion. Furthermore, loop-carried flow dependence edges from a vertex to itself will be omitted, as such edges are irrelevant for the computation of slices.

For each edge $i \rightarrow_{\text{CFG}} j$ in the CFG:

$$R^0_C(i) = R^0_C(i) \cup \{v \mid v \in R^0_C(j), v \notin \text{DEF}(i)\} \cup \{v \mid v \in \text{REF}(i), \text{DEF}(i) \cap R^0_C(j) \neq \emptyset\}$$
$$S^0_C = \{i \mid (\text{DEF}(i) \cap R^0_C(j)) \neq \emptyset, i \rightarrow_{\text{CFG}} j\}$$

**Fig. 5.** Equations for determining *directly* relevant variables and statements

## 3. Methods for static slicing

### 3.1 *Basic algorithms*

In this section, we will study basic algorithms for static slicing of structured, single-procedure programs with scalar variables. These algorithms essentially compute the same information, but in different ways.

#### 3.1.1 *Dataflow equations*

Weiser's original definition of program slicing [3] is based on the iterative solution of dataflow equations[5]. 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 properties: (i) $S$ must be a valid program, and (ii) 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.

Weiser describes an iterative algorithm for computing approximations of statement-minimal slices. It is important to realize that this algorithm uses *two* distinct 'layers' of iteration. These can be characterized as follows:

(1) Tracing transitive data dependences. This requires iteration in the presence of loops.
(2) Tracing control dependences, causing the inclusion in the slice of certain control predicates. For each such predicate, step 1 is repeated to include the statements it is dependent upon.

The algorithm determines consecutive sets of *relevant variables* from which sets of *relevant statements* are derived; the computed slice is defined as the fixpoint of the latter set. First, the *directly relevant variables* are determined: this is an instance of step 1 of the iterative process outlined above. The set of directly relevant variables at node $i$ in the CFG is denoted $R^0_C(i)$. The iteration starts with the initial values $R^0_C(n) = V$, and $R^0_C(m) = \emptyset$ for any node $m \neq n$. Figure 5 shows a set

[5] Weiser's definition of branch statements with indirect relevance to a slice contains an error [54]. In the present paper, the modified definition proposed in [55] is followed. However, we do not agree with the statement in [55] that 'It is not clear how Weiser's algorithm deals with loops'.

$$B_C^k = \{b \mid \exists i \in S_C^k, i \in \textsc{Infl}(b)\}$$
$$R_C^{k+1}(i) = R_C^k(i) \cup \bigcup_{b \in B_C^k} R^0_{(b,\textsc{Ref}(b))}(i)$$
$$S_C^{k+1} = B_C^k \cup \{i \mid \textsc{Def}(i) \cap R_C^{k+1}(j) \neq \emptyset, i \rightarrow_{\text{CFG}} j\}$$

**Fig. 6.** Equations for determining *indirectly* relevant variables and statements

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

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

of equations that define how the set of relevant variables at the *end* $j$ of a CFG edge $i \rightarrow_{\text{CFG}} j$ affects the set of relevant variables at the *beginning* $i$ of that edge. The least fixed point of this process is the set of directly relevant variables at node $i$. From $R_C^0$, a set of *directly relevant statements*, $S_C^0$, is derived. Figure 5 shows how $S_C^0$ is defined as the set of all nodes $i$ that define a variable $v$ that is a relevant at a CFG-successor of $i$.

As mentioned, the second 'layer' of iteration in Weiser's algorithm consists of taking control dependences into account. 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. To this end, the *range of influence* $\textsc{Infl}(b)$ of a branch statement $b$ is defined as the set of statements control dependent[6] on $b$. Figure 6 shows a definition of the branch statements $B_C^k$ that are indirectly relevant due to the influence they have on nodes $i$ in $S_C^k$. Next, the sets of *indirectly relevant variables* $R_C^{k+1}(i)$ are determined. In addition to the variables in $R_C^k(i)$, $R_C^{k+1}(i)$ contains variables that are relevant because they have a transitive data dependence on statements in $B_C^k$. This is determined by performing the first type of iteration again (i.e. tracing transitive data dependences) with respect to a set of criteria $(b, \textsc{Ref}(b))$ where $b$ is a branch statement in $B_C^k$ (see Fig. 6). Figure 6 also shows a definition of the sets $S_C^{k+1}$ of *indirectly relevant statements* in iteration $k + 1$. This set consists of the the nodes in $B_C^k$ together with the nodes $i$ that define a variable that is $R_C^{k+1}$-relevant to a CFG-successor $j$.

[6] Actually, Weiser defines $\textsc{Infl}(b)$ to be the set of statements *transitively* control dependent on $b$. The definition we use here produces equivalent slices, but is more efficient.

$$\begin{array}{ll|lll|l}
\lambda_\epsilon & = \emptyset & \lambda_{S_1;S_2} & = & \lambda_{S_1} \cup (\rho_{S_1} \cdot \lambda_{S_2}) & \lambda_{v:=e} = \textsc{Vars}(e) \times \{e\} \\
\mu_\epsilon & = \emptyset & \mu_{S_1;S_2} & = & (\mu_{S_1} \cdot \rho_{S_2}) \cup \mu_{S_2} & \mu_{v:=e} = \{(e,v)\} \\
\rho_\epsilon & = \textsc{Id} & \rho_{S_1;S_2} & = & \rho_{S_1} \cdot \rho_{S_2} & \rho_{v:=e} = (\textsc{Vars}(e) \times \{v\}) \cup (\textsc{Id} - (v,v))
\end{array}$$

$$\begin{array}{ll}
\lambda_{\mathbf{if}\, e\, \mathbf{then}\, S_1\, \mathbf{else}\, S_2} & = (\textsc{Vars}(e) \times \{e\}) \cup \lambda_{S_1} \cup \lambda_{S_2} \\
\mu_{\mathbf{if}\, e\, \mathbf{then}\, S_1\, \mathbf{else}\, S_2} & = (\{e\} \times (\textsc{Defs}(S_1) \cup \textsc{Defs}(S_2))) \cup \mu_{S_1} \cup \mu_{S_2} \\
\rho_{\mathbf{if}\, e\, \mathbf{then}\, S_1\, \mathbf{else}\, S_2} & = (\textsc{Vars}(e) \times (\textsc{Defs}(S_1) \cup \textsc{Defs}(S_2))) \cup \rho_{S_1} \cup \rho_{S_2} \\
\\
\lambda_{\mathbf{while}\, e\, \mathbf{do}\, S} & = \rho_S^\star \cdot ((\textsc{Vars}(e) \times \{e\}) \cup \lambda_S) \\
\mu_{\mathbf{while}\, e\, \mathbf{do}\, S} & = (\{e\} \times \textsc{Defs}(S)) \cup \mu_S \cdot \rho_S^\star \cdot ((\textsc{Vars}(e) \times \textsc{Defs}(S)) \cup \textsc{Id}) \\
\rho_{\mathbf{while}\, e\, \mathbf{do}\, S} & = \rho_S^\star \cdot ((\textsc{Vars}(e) \times \textsc{Defs}(S)) \cup \textsc{Id})
\end{array}$$

**Fig. 7.** Definition of information-flow relations

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, consider slicing the program of Fig. 1 (a) with respect to 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 Fig. 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 Fig. 1 (b) apart from the fact that the output statement `write(product)` is not contained in the slice.

Lyle [56] presents a modified version of Weiser's slicing algorithm. Apart from some minor changes in terminology, this algorithm is essentially the same as Weiser's [3].

Hausler [57] restates Weiser's algorithm in a functional style. For each type of statement (empty statement, assignment, statement composition, **if**, and **while**) he defines two functions $\delta$ and $\alpha$. Roughly speaking, these functions express how a statement transforms the set of relevant variables $R_C^i$, and the set of relevant statements $S_C^i$, respectively. The functions $\delta$ and $\alpha$ are defined in a compositional manner. For empty statements and assignments, $\delta$ and $\alpha$ can be derived from the statement in a syntax-directed manner. The $\delta$ and $\alpha$ functions for statement sequences and **if**

| Expression number[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 Fig. 1 (a).

**Fig. 8.** Information-flow relation $\mu$ for the example program of Fig. 1 (a)

statements, can be inferred from the $\delta$ and $\alpha$ functions for their components, respectively. The functions for a **while** statement are obtained by effectively transforming it into an infinite sequence of **if** statements.

### 3.1.2 *Information-flow relations*

Bergeretti and Carré [7] define a number of *information-flow relations* that 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 $\lambda_S$, $\mu_S$, and $\rho_S$ 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. This yields the slice with respect to the final value of $v$.

Information-flow relations are computed in a syntax-directed, bottom-up manner. For an empty statement, the relations $\lambda_S$ and $\mu_S$ are empty, and $\rho_S$ is the identity. For an assignment $v := e$, $\lambda_S$ contains $(v', e)$ for all variables $v'$ that occur in $e$, $\mu_S$ consists of $(e, v)$, and $\rho_S$ contains $(v', v)$ for all variables $v'$ that occur in $e$ as well as $(v'', v'')$ for all variables $v'' \neq v$. Figure 7 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, $\epsilon$ denotes an empty statement, '$\cdot$' relational join[7], $\textsc{Id}$ the identity relation, $\textsc{Vars}(e)$ the set of variables occurring in expression $e$, and $\textsc{Defs}(S)$ 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 $\rho$.

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 $\lambda_S$ in a way that is similar to the method for computing static backward slices from the $\mu_S$ relation.

Figure 8 shows the information-flow relation $\mu$ for the (entire) program of Fig. 1 (a)[8]. From this relation it follows that the set of expressions that 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 that do not contain expressions in this set, i.e. both assignments to `sum` and both `write` statements. The result is identical to the slice computed by Weiser's algorithm (see Section 3.1.1).

[7] The join of two relations $R_1$ and $R_2$ contains all pairs $(e_1, e_3)$ for which there exists an $e_2$ such that $(e_1, e_2) \in R_1$ and $(e_2, e_3) \in R_2$.

[8] Bergeretti an 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.

### 3.1.3 *Dependence graph based approaches*

Ottenstein and Ottenstein [4] were the first of many to define slicing as a reachability problem in a dependence graph representation of a program. They use the PDG [5, 6] for static slicing of single-procedure programs.

In 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 $v$ (the fine-grained PDGs of Jackson and Rollins, discussed below, are an exception here). However, we consider this difference to be insignificant; in Section 3.6.2, it will be discussed 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 that can reach $v$. 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 Ottenstein and Ottenstein [4] shows considerably more detail than that by Horwitz *et al.* [36]. 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.

In Fig. 4 the PDG of the program of Fig. 1 (a) was shown. Shading is used to indicate the vertices in the slice w.r.t. `write(product)`.

Jackson and Rollins [32] introduce a variation on the PDG that is distinguished by fine-grained dependences between individual variables defined or used at program points. An advantage of this approach is that it allows for slicing criteria that are more detailed than those of the previously discussed PDG-based algorithm. A slice w.r.t. any variable that is either defined or used at a program point can be extracted directly from the dependence graph. This enables one to determine more accurately which variables are responsible for the inclusion of a particular statement in a slice.

Each vertex consists of a *box* that contains a separate *port* for each variable defined at that program point, as well as for each variable used at that point. Dependence relations between variables used at a program point $p$, and variables defined at $p$ are represented by *internal* dependence edges *inside* the box for $p$. Data dependences between statements are defined in the usual way, in terms of reaching definitions. Control dependences between statements, however, are modelled as mock data dependences. To this end, each box has an $\epsilon$ port that represents the 'execution of' the associated statement. Control predicates are assumed to define a temporary value that is represented by a $\tau$ port. If a statement with box $p$ is control dependent on a statement with box $q$, this is modelled by a dependence edge from $p$'s $\tau$ port to $q$'s $\epsilon$ port. Finally, dependences on constant values and input values are represented by $\gamma$ ports—the role of these ports is irrelevant for the present discussion.

Jackson and Rollins generalize the traditional notion of a slicing criterion to a pair (*source, sink*), where *source* is a set of definition ports and *sink* a set of use ports. Slicing is generalized to *chopping*: determining the subset of the program's statements that cause influences of *source* elements on *sink* elements. Conceptually, chops can be computed by solving a reachability problem in a modular dependence graph. However, Jackson and Rollins formally define their algorithm in a purely relational fashion, as a number of relations between ports; a description of this is outside the scope of this paper. It is argued that conventional notions of backward and forward slicing can be expressed in terms of chopping.

## 3.2 *Procedures*

The main problem posed by interprocedural static slicing is that, in order to compute accurate slices, the call-return structure of interprocedural execution paths must be taken into account. Simple algorithms that include statements in a slice by traversing (some representation of) the program in a single pass have the property that they consider *infeasible* execution paths, causing slices to become larger than necessary. Several solutions to this problem, often referred to as the 'calling-context' problem, will be discussed below.

### 3.2.1 *Dataflow equations*

Weiser's approach for interprocedural static slicing [3, 54] involves three separate tasks.

- First, interprocedural *summary information* is computed, using previously developed techniques [58]. For each procedure $P$, a set $\textsc{Mod}(P)$ of variables that may be modified by $P$ is computed, and a set $\textsc{Use}(P)$ of variables that may be used by $P$. In both cases, the effects of procedures transitively called by $P$ are taken into account.
- The second component of Weiser's algorithm is an *intra*procedural slicing algorithm. This algorithm was discussed previously in Section 3.1.1. However, it is slightly extended in order to determine the effect of call-statements on the sets of relevant variables and statements that are computed. This is accomplished using the summary information computed in step (1). A call to procedure $P$ is treated as a conditional assignment statement '**if** ⟨SomePredicate⟩ **then** $\textsc{Mod}(P) := \textsc{Use}(P)$' where actual parameters are substituted for formal parameters [54]. Worst-case assumptions have to be made when a program calls external procedures, and the source-code is unavailable.
- The third part is the actual interprocedural slicing algorithm that iteratively generates new slicing criteria with respect to which intraprocedural slices are computed in step (2). For each procedure $P$, new criteria are generated for (i) procedures $Q$ called by $P$, and (ii) procedures $R$ that call $P$. 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 $P$ 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 $P$ that is in the scope of $R$ (actuals are substituted for formals).

Weiser formalizes the generation of new criteria by way of functions $\textsc{Up}(\mathcal{S})$ and $\textsc{Down}(\mathcal{S})$ that 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 set of *all* criteria with respect to which intraprocedural slices are computed consists of the transitive and reflexive closure of the $\textsc{Up}$ and $\textsc{Down}$ relations; this is denoted $(\textsc{Up} \cup \textsc{Down})^*$. Thus, for an initial criterion $C$, slices will be computed for all criteria in the set $(\textsc{Up} \cup \textsc{Down})^*(\{C\})$.

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

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

**Fig. 9.** A program where procedure P is sliced $n$ times by Weiser's algorithm for criterion (L, {z})

Weiser determines the criteria in this set 'on demand' [54]: the generation of new criteria in step (3) and the computation of intraprocedural slices in step (2) are intermixed; the iteration stops when no new criteria are generated. Although the number of intraprocedural slices computed in step (2) could be reduced by combining 'similar' criteria (e.g. replacing two criteria $(n, V_1)$ and $(n, V_2)$ by a single criterion $(n, V_1 \cup V_2)$), Weiser writes that 'no speed-up tricks have been implemented' [3, page 355, col.2]. In fact, one would expect that such speed-up tricks would affect the performance of his algorithm dramatically. The main issue is that 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 relies on *slicing* these procedures. In the course of doing this, new variables may become relevant at previously encountered call sites, and new call sites may be encountered. Consider for example, the program shown in Fig. 9. In the subsequent discussion, $L$ denotes the program point at statement `write(z)` and $M$ the program point at 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 $i^{\text{th}}$ iteration, variables $x_1, \cdots, x_i$ will be relevant at the call site, causing the

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

    procedure P(v, w, x, y);
(5)   x := v;
(6)   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)

**Fig. 10.** (a) Example program. (b) Weiser's slice with respect to criterion (4, {d}). (c) A slice with respect to the same criterion computed by the Horwitz–Reps–Binkley algorithm

```
      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
```

**Fig. 11.** Example of a multi-procedure program

inclusion of criterion $(M, \{y_1, \cdots, y_l\})$ in Down(Main). If no precaution is taken to combine the criteria in Down(Main), procedure P will be sliced $n$ times.

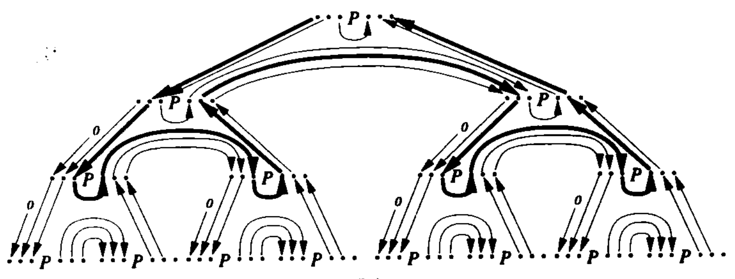
The fact that Weiser's algorithm does not take into account *which* output parameters are dependent on *which* input parameters is a source of imprecision. Figure 10 (a) shows an example program that manifests this problem. For criterion $(4, \{d\})$, Weiser's interprocedural slicing algorithm [3] will compute the slice shown in Fig. 10 (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 Fig. 10 (c).

Horwitz *et al.* [36] report 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 includes *infeasible* execution paths that enter $Q$ from $P$ and exit $Q$ to a different procedure. Traversal of such paths gives rise to inaccurate slices.

Figure 11 shows a program that exhibits the calling-context problem. For example, assume that a slice is to be computed w.r.t. criterion (10, product). Using summary information to approximate the effect of the calls, the initial approximation of the slice will consist of the entire main procedure except lines 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 Mod(Add) = {a}, Use(Add) = {a, b}, Mod(Multiply) = {c}, and Use(Multiply) = {c, d}. As the initial criterion is in the main program, we have that Up{(10, product)} = $\emptyset$, and that Down{(10, product)} contains the criteria (11, {a}) and (17, {c, d}). The result of slicing procedure Add for criterion (11, {a}) and procedure Multiply for criterion (17, {c, d}) will be the inclusion of these procedures in their entirety. Note that the calls to Add at lines 15 and 16 cause the generation of a new criterion (11, {a, b}) and thus re-slicing of procedure Add. It can now

```
    program P³(x₁,x₂,x₃);
    begin
      t := 0;
      P³(x₂,x₃,t);
      P³(x₂,x₃,t);
(L)   x₁ := x₁ + 1
    end;
```

**(a)**



**(b)**

**Fig. 12.** (a) Example program. (b) Exponentially long path traversed by the Hwang–Du–Chou algorithm for interprocedural static slicing for criterion $(L, x_3)$

be seen that the example program 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, {`sum`, `i`}) that is generated will cause the inclusion of lines 6 and 3 in the slice. Consequently, the slice consists of the entire program.

It is our 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 (transitively) call the procedure containing the initial criterion[9]. Once this is done, *only* DOWN sets need to be computed. For an initial criterion $C$, this corresponds to determining the set of criteria $\text{DOWN}^*(\text{UP}^*(\{C\}))$, and computing the intraprocedural slices with respect to each of these criteria. Reps [60] 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.

Hwang *et al.* [61] propose an iterative solution for interprocedural static slicing based on replacing (recursive) calls by instances of the procedure body. From a conceptual point of view, each iteration comprises the following two steps. First, procedure calls are inlined, substituting actual parameters for formal parameters. Then, the slice is re-computed, where any remaining procedure call is treated as if it were an empty statement (i.e. it is assumed to have no effect on the flow dependences between its surrounding statements). This iterative process terminates when the resulting slice is identical to the slice computed in the previous iteration, i.e. until a fixed point is reached. It is assumed that some mapping is maintained between the statements in the various expanded versions of the program, and in the original program.

The approach of Hwang *et al.* does not suffer from the calling context problem because expansion of recursive calls does not lead to considering infeasible execution paths. However, Reps [62, 63] 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. An example of such a program is shown in Fig. 12 (a). Figure 12 (b) shows the exponentially long path that is effectively traversed by the Hwang–Du–Chou algorithm.

[9] A similar observation was made by Jiang *et al.* [59]. However, they do not explain that this approach only works when a call to procedure $P$ is treated as a conditional assignment **if** ⟨SomePredicate⟩ **then** $\text{MOD}(P) := \text{USE}(P)$.

### 3.2.2 *Information-flow relations*

Bergeretti and Carré [7] explain how the effect of procedure calls can be approximated in the absence of recursion. 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 $\mu$ 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 parameters 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 modelled by replacing it with a fictitious expression containing all actual parameters.

Note that the computed slices are not truly interprocedural since slices are not extended to procedures other than the one containing the slicing criterion, as is done in the algorithm of Section 3.2.1.

For the example program of Fig. 11, 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 *et al.* [36] present an algorithm for computing precise interprocedural static slices, which consists of the following three components:

(1) The *System Dependence Graph* (SDG), a graph representation for multi-procedure programs.
(2) The computation of interprocedural summary information. This takes the form of precise dependence relations between the input and output parameters of each procedure call, and is explicitly present in the SDG in the form of *summary edges*.
(3) A two-pass algorithm for extracting interprocedural slices from an SDG.

We will begin with a brief overview of SDGs. In the discussion that follows it is important to realize that parameter passing by value-result[10] is modelled 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 that 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 that 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

[10] The Horwitz–Reps–Binkley algorithm [36] is also suitable for call-by-reference parameter passing provided that aliases are resolved. To this end, two approaches are proposed: transformation of the original program into an equivalent alias-free program, or the use of a generalized flow dependence notion that takes possible aliasing patterns into account. The first approach yields more precise slices, whereas the second one—further exploited by Binkley [64]—is more efficient. For a discussion of parameter passing mechanisms the reader is referred to [65, Section 7.5].

model copying of formal parameters to/from temporary variables[11]. 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) *summary edges* that represent *transitive inter*procedural data dependences.

The second part of the Horwitz–Reps–Binkley algorithm consists of the computation of summary edges between (vertices for) input and output parameters of a procedure. The presence of such an edge reflects the fact that the incoming value of the input parameter may be used in obtaining the outgoing value of the output parameter. The Horwitz–Reps–Binkley algorithm determines summary edges by constructing an attribute grammar that models the calling relationships between the procedures (as in a call graph). Then, the subordinate characteristic graph for this grammar is computed. For each procedure in the program, this graph contains edges that correspond to precise transitive flow dependences between its input and output parameters. The summary edges of the subordinate characteristic graph are copied to the appropriate places at each call site in the SDG. Details of the Horwitz–Reps–Binkley algorithm for determining summary edges are outside the scope of this survey—for details, the reader is referred to [36]. Recently, more efficient algorithms for determining summary edges have been presented [67, 63, 50]; these algorithms are discussed below.

The third phase of the Horwitz–Reps–Binkley algorithm consists of a two-pass traversal of the SDG. The summary edges of an SDG serve to circumvent the calling context problem. Assume that slicing starts at some 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. The second phase determines the remaining vertices in the slice by descending into all previously side-stepped calls.

Figure 13 shows the SDG for the program of Fig. 11, 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 arrows 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 Horwitz–Reps–Binkley algorithm [36] 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. Two approaches are proposed for transforming such a non-executable slice into an executable program. First, several variants of a procedure

[11] Using interprocedural data flow analysis [66], the sets of variables that 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.

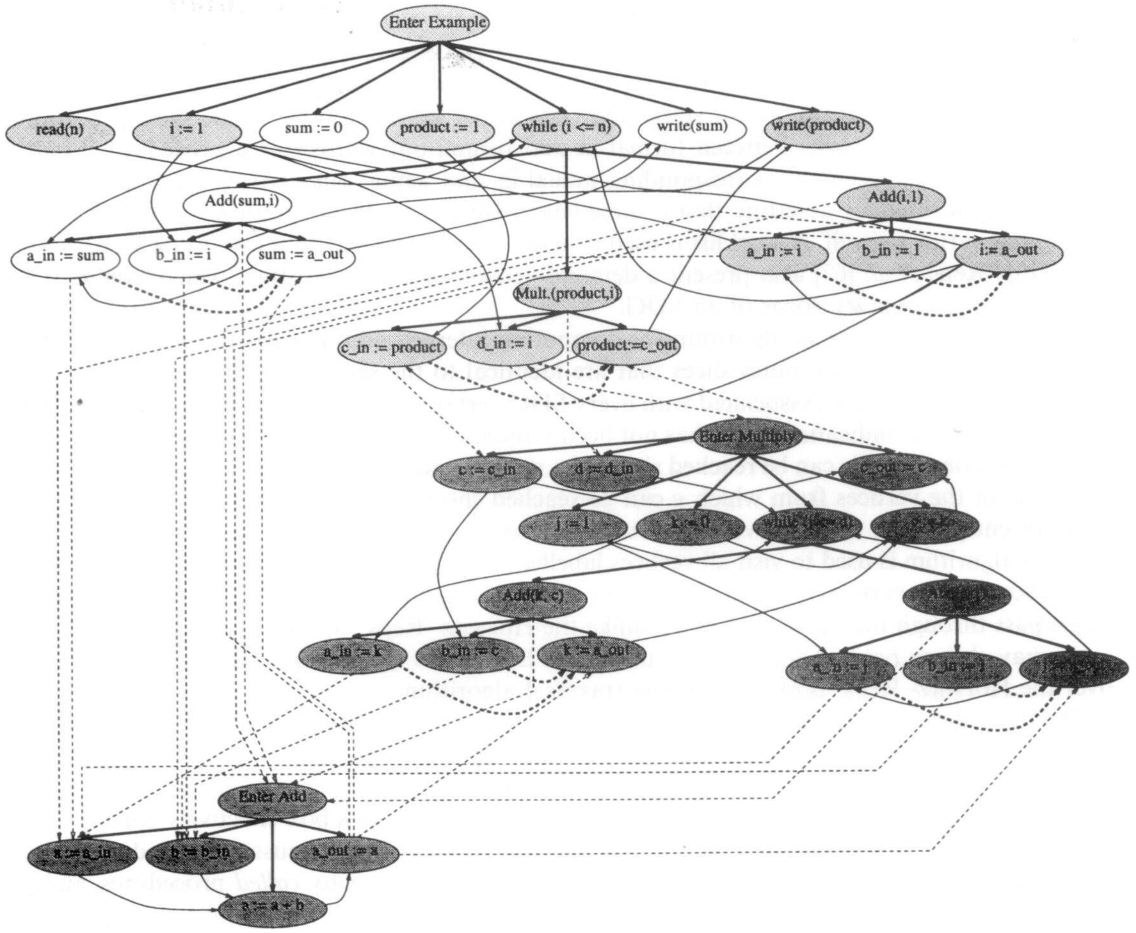


**Fig. 13.** SDG of the program in Fig. 11

may be incorporated in a slice [36]; this 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 that occur in the slice. In addition, all vertices on which the added vertices are dependent must be added to the slice as well. This second approach is pursued by Binkley [68]. Clearly it yields larger slices than the first one.

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.* [67, 63] proposed a new algorithm for computing the summary edges of an SDG, which is asymptotically more efficient than the Horwitz–Reps–Binkley algorithm [36] (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. Reps *et al.* [67] also present a demand version of their algorithm, which *incrementally* determines the summary edges of an SDG.

Lakhotia [69] 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 Horwitz–Reps–Binkley algorithm. Associated with every SDG vertex $v$ is a three-valued tag; possible values for this tag are: '$\perp$' 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 '$\beta$' 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 labelled $\top$. A worklist algorithm is used to visit all vertices labelled $\top$ before visiting any vertex labelled $\beta$. When this process ends, vertices labelled either $\top$ or $\beta$ are in the slice. Lakhotia's algorithm performs a single pass through the SDG. However, unlike the Horwitz–Reps–Binkley algorithm, 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.

The dependence graph model of Jackson and Rollins [32] (see Section 3.1.3) is 'modular', in the sense that a single box is used for each procedure call. Instead of linking the individual dependence graphs for the procedures of the program, Jackson and Rollins represent procedure calls in a more abstract fashion: the internal dependence edges inside a procedure's box effectively correspond to the summary edges of Horwitz *et al.* [36, 63]. Unlike the previously discussed methods, this algorithm side-steps the calling context problem by only extending slices to *called* procedures, and not to *calling* procedures (unless explicitly requested by the user). Here, 'extending a slice to a called procedure' involves slicing the (separate) dependence graph for that procedure with respect to the appropriate ports of its exit node (i.e. corresponding to the ports at the point of call that occur in the slice).

Whereas for simple statements the internal dependence edges between ports of the associated box in the dependence graph can be computed in a simple syntax-directed manner, a more elaborate scheme is required for procedures. In the absence of recursion, the internal summary dependence edges for a procedure are derived from the dependences inside and between the boxes for the statements that constitute the procedure body. For recursive procedures, Jackson and Rollins briefly discuss a simple iterative scheme for determining internal dependence edges, and state that their algorithm is essentially an adaptation of the solution presented by Ernst [50] (see

**Fig. 14.** (a) Program with unstructured control flow, (b) PDG for program of (a), (c) incorrect slice w.r.t. statement `write(product)`, (d) Augmented PDG for program of (a), (e) correct slice w.r.t. statement `write(product)`

```
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)
```

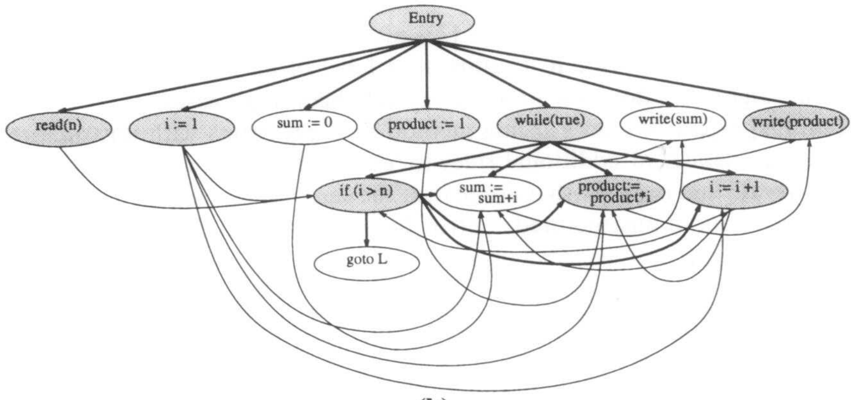
(a)

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

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

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

write(product)
```

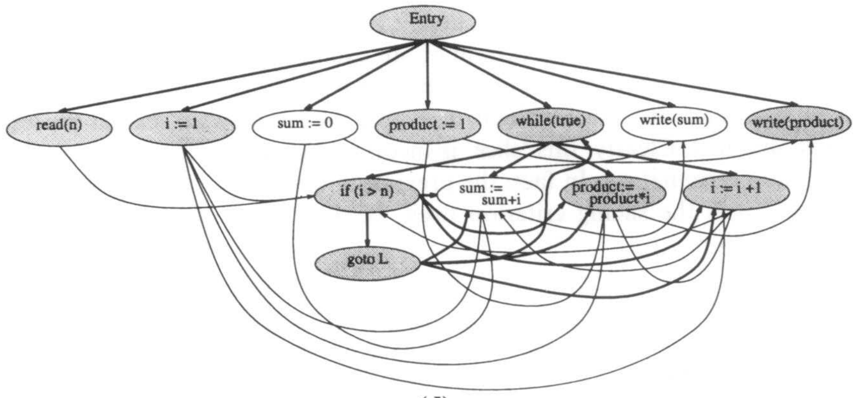
(c)

```
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)

Section 6). The essence of their scheme is that the internal dependence edges for non-recursive calls are determined in the manner sketched above, and that there are initially no internal dependence edges for calls in a recursive cycle. In each subsequent step, the transitive dependences between the input parameters and the output parameters of a recursive procedure are recomputed by slicing in a graph that contains the summary edges determined in the previous cycle. Then, summary edges are added to the graph for those dependences that did not occur in the previous cycle. This iterative process terminates when no more additional transitive dependences can be found.

## 3.3 *Unstructured control flow*

### 3.3.1 *Dataflow equations*

Lyle [56] reports that (his version of) Weiser's algorithm for static slicing yields incorrect slices in the presence of unstructured control flow: the behaviour of the slice is not necessarily a projection of the behaviour of the program. He presents a conservative solution for dealing with **goto** statements: any **goto** that has a non-empty set of relevant variables associated with it is included in the slice.

Gallagher [70] and Gallagher and Lyle [27] 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[12]. Agrawal [47] shows that this algorithm does not produce correct slices in all cases.

Jiang *et al.* [59] extend Weiser's slicing method to C programs with unstructured control flow. They introduce a number of additional rules to 'collect' the unstructured control flow statements such as **goto**, **break**, and **continue** that are part of the slice. Unfortunately, no formal justification is given for the treatment of unstructured control flow constructs in [59]. Agrawal [47] shows that this algorithm may also produce incorrect slices.

### 3.3.2 *Dependence graphs*

Ball and Horwitz [45, 46] and Choi and Ferrante [48] 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, Fig. 14 (a) shows a variant of our example program, which uses a **goto** statement. Figure 14 (b) shows the PDG for this program. The vertices that have a transitive dependence on statement `write(product)` are highlighted. Figure 14 (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**.

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

The solution of [45, 46] and the first solution presented in [48] 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'. In constructing the (augmented) PDG, data dependences are computed using the (original) CFG, and control dependences are computed using the ACFG. Slicing is defined in the usual way, 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 14 (d) shows the augmented PDG for the program of Fig. 14 (a); vertices from which the vertex labelled `write(product)` can be reached are indicated by shading. The (correct) slice corresponding to these vertices is shown in Fig. 14 (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 of it, however, is that slices may include **goto** statements that do not occur in the original program.

Yet another PDG-based method for slicing programs with unstructured control flow was recently proposed by Agrawal [47]. Unlike the methods by Ball and Horwitz [45, 46] and Choi and Ferrante [48], Agrawal uses *unmodified* PDGs. 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.

Agrawal's key observation is that an unconditional jump statement $J$ should be included in the slice if and only if the immediate postdominator of $J$ that is included in the slice differs from the immediate lexical successor of $J$ that is included in the slice. Here, a statement $S'$ is a *lexical successor* of a statement $S$ if $S$ textually precedes $S'$ in the program[13]. 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

[13] As Agrawal observes, this notion is equivalent to the non-executable edges in the augmented control flow graphs used by Ball and Horwitz, and Choi and Ferrante.

$S_3$ 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 the jump statement 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 is necessary because adding a jump (and the statements it depends 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 identical to those computed by the Ball–Horwitz and Choi–Ferrante algorithms.

Agrawal's algorithm [47] 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 Fig. 14 (a) because it is control dependent on the (included) predicate of the **if** statement.

## 3.4 *Composite data types and pointers*

Lyle [56] proposes a conservative solution to the problem of static slicing in the presence of arrays. 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 [4] 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—a phenomenon commonly called *aliasing*. Aliasing complicates the computation of slices because the notion of flow dependence depends on which variables are (potential) aliases. Even in the *intra*procedural case, the problem of determining potential aliases in the presence of multiple-level pointers is an $\mathcal{NP}$-hard problem [51]. However, slices may be computed using conservative approximations of data dependences that are based on approximate alias information. Conservative algorithms for determining potential aliases were presented by Landi and Ryder [52], and Choi *et al.* [71].

Horwitz *et al.* [72] present a slightly different approach for computing flow dependences in the presence of pointers. Instead of defining (approximate) flow dependences in terms of definitions and uses of variables that are potentially aliased, the notion of flow dependence is defined in terms of potential definitions and uses of *abstract memory locations*. An algorithm is presented that computes approximations of the memory layouts that may occur at each program point during execution.

The PDG-based static slicing algorithm proposed by Agrawal *et al.* [73] implements a similar idea to deal with both composite variables and pointers. Their solution consists of determining reaching definitions for a scalar variable $v$ at node $n$ in the flowgraph by finding all paths from nodes corresponding to a definition of $v$ to $n$ that do not contain other definitions of $v$. When composite data types and pointers are considered, definitions involve *l-valued expressions* rather than variables. An l-valued expression is any expression that may occur as the left-hand side of an assignment. Agrawal *et al.* present a new definition of reaching definitions that is based on the layout of memory locations potentially denoted by l-valued expressions. 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 reach a use, the situation becomes more complex when composite data types 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 some array $a$ is used at $e_2$, and some element $a[i]$ of $a$ is defined at $e_1$.

Given these concepts, an extended reaching definition function is defined that 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 that have not been defined yet in the case of Partial Intersections.

Lyle and Binkley [74] present an approach for slicing in the presence of pointers that is based on a variation of symbolic execution. Their algorithm consists of two phases. First, all CFG nodes are determined that introduce addresses (either due to a use of the C '&' operator, or due to the dynamic allocation of a new object). These addresses are propagated through the CFG yielding a set of address values for each pointer at each program point. A number of propagation rules defines how addresses are propagated by assignments statements[14]. In the second phase, the information collected in the first phase is used to determine which statements should be included in a slice. This second phase is essentially a generalization of Lyle's slicing algorithm [56].

Jiang *et al.* [59] present an algorithm for slicing C programs with pointers and arrays that is based on Weihl's notion of *dummy variables* [75]. The basic idea is that for each pointer $p$, the dummy variable $(1)p$ denotes the value pointed to by $p$, and for each variable $x$, $(-1)x$ denotes the address of $q$. Jiang *et al.* define data dependences in the usual way, in terms of definitions and uses of (dummy) variables. Unfortunately, this approach appears to be flawed[15]. Figure 15 shows an example program, the DEF, REF, and $R_C^0$ sets for each statement, and the incorrect slice computed

[14] In their definitions, Lyle and Binkley only address straight-line code, and argue that control-dependence issues are 'orthogonal' to the data-dependence issues raised by pointers.

[15] The counterexample of Fig. 15 was provided by Susan Horwitz.

```
(1) p = &x;
(2) *p = 2;
(3) q = p;
(4) write(*q)
```

(a)

| # | DEF | REF | $R_C^0$ |
|---|---|---|---|
| 1 | {p} | {(-1)x} | ∅ |
| 2 | {(1)p} | {p} | {p, (1)q} |
| 3 | {q} | {p} | {p, (1)q} |
| 4 | ∅ | {q, (1)q} | {q, (1)q} |

(b)

```
(1) p = &x;
(2)        ;
(3) q = p;
(4)
```

(c)

**Fig. 15.** (a) Example program. (b) Defined variables, used variables, and relevant variables for this program. (c) Incorrect slice w.r.t. criterion $C = (4, \{q, (1)q\})$

for criterion $C = (4, q, (1)q)$. The second statement is incorrectly omitted because it does not define any variable that is relevant at statement 3.

## 3.5 *Concurrency*

Cheng [76] 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 usual PDG edges, PDNs 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 program point 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 [77] for the construction of PDNs. The set of weak control dependences is a superset of the set of control dependences, the difference being the fact that weak control dependences exist between the control predicate of a loop and the statements that follow it. For example, the statements on lines 9 and 10 of the program of Fig. 1 (a) are weakly (but not strongly) control dependent on the control predicate on line 5.

## 3.6 *Comparison*

### 3.6.1 *Overview*

In this section, the static slicing methods that were presented earlier are compared and classified. The section is organized as follows. Section 3.6.1 summarizes the problems that are addressed in

**Table 2.** Overview of static slicing methods

| *Reference* | *Computation method*[a] | *Interprocedural solution* | *Control flow*[b] | *Data types*[c] | *Concurrency* |
|---|---|---|---|---|---|
| Weiser [3, 55] | D | yes | S | S | no |
| Lyle [56] | D | no | A | S, A | no |
| Gallagher, Lyle [70, 27] | D | no | A[d] | S | no |
| Jiang *et al.* [59] | D | yes | A[d] | S, A, P[e] | no |
| Lyle, Binkley [74] | D | no | S[f] | S, P | no |
| Hausler [57] | F | no | S | S | no |
| Bergeretti, Carré [7] | I | yes[g] | S | S | no |
| Ottenstein [4] | G | no | S | S, A | no |
| Horwitz *et al.* [26, 42, 43] | G | no | S | S | no |
| Horwitz *et al.* [36] | G | yes | S | S | no |
| Binkley [64] | G | yes[h] | S | S | no |
| Binkley [78] | G | yes[i] | S | S | no |
| Jackson, Rollins [32, 33] | G | yes | S | S | no |
| Reps et al. [67, 63] | G | yes | S | S | no |
| Lakhotia [69] | G | yes | S | S | no |
| Agrawal *et al.* [73] | G | no | S | S, A, P | no |
| Ball, Horwitz [45, 46] | G | no | A | S | no |
| Choi, Ferrante [48] | G | no | A | S | no |
| Agrawal [47] | G | no | A | S | no |
| Cheng [76] | G | no | S | S | yes |
| Ernst [50] | O | yes | A | S, A, P | no |
| Field *et al.* [17, 18, 79] | R | no | S | S, P | no |

[a] D = dataflow equations, F = functional/denotational semantics, I = information-flow relations, G = reachability in a dependence graph, O = dependence graphs in combination with optimization techniques (see Section 6). R = dependence tracking in term graph rewriting systems (see Section 6).
[b] S = structured, A = arbitrary.
[c] S = scalar variables, A = arrays/records, P = pointers.
[d] Solution incorrect (see [47]).
[e] Solution incorrect (see Section 3.4).
[f] Only straight-line code is considered.
[g] Non-recursive procedures only.
[h] Takes parameter aliasing into account.
[i] Produces slices that are executable programs.

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 possibilities for combining algorithms that deal with different problems are discussed.

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 data types under consideration, and (iv) whether or not concurrency 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 Horwitz–Reps–Binkley interprocedural slicing algorithm [36] could in principle be combined with any of the dependence graph based slicing methods for dealing with unstructured control flow [46–48]. Possibilities for such combinations are investigated to some extent in Section 3.6.4. The work by Ernst [50] and by Field *et al.* [17, 18, 79] that occurs in Table 2 relies on substantially different techniques than those used for the static slicing algorithms discussed previously, and will therefore be studied separately in Section 6.

Kamkar [44] distinguishes between methods for computing slices that are executable programs, and those for computing slices that consist of a set of 'relevant' statements. We agree with the observation by Horwitz *et al.* [36], that for *static* slicing of *single-procedure* programs this is merely a matter of presentation. However, for multi-procedure programs, the distinction *is* significant, as was remarked in Section 3.2.3. Nevertheless, we believe that the distinction between executable and non-executable interprocedural slices can be ignored in this case as well, because the problems are strongly related: Binkley [68] describes how precise executable interprocedural static slices can be obtained from the non-executable interprocedural slices computed by the algorithm of Horwitz *et al.* [36].

A final remark here concerns I/O statements. The slices computed by Weiser's algorithm [3] and the algorithm by Bergeretti and Carré [7] never contain output statements because: (i) the Def set of an output statement is empty so that no other statement is data dependent on it; and (ii) no statement is control dependent on an output statement. Horwitz and Reps [40] 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 in the criterion.

## 3.6.2 *Accuracy*

Although the problem of determining statement-minimal slices is undecidable in general, some algorithms compute better approximations of statement-minimal slices than other algorithms. For convenience, we will call an algorithm 'inaccurate' if another algorithm computes a smaller slice for the same criterion.

An issue that at first glance seems to complicate the comparison of the static slicing methods discussed previously is the fact that some methods allow more general slicing criteria than others. 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, with the exception of the 'modular' PDGs of Jackson and Rollins [32], the slicing criteria of PDG-based slicing methods effectively correspond to a pair $(s, \text{Vars}(s))$, where $s$ is a statement and $\text{Vars}(s)$ 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 definitions that reach a variable in $V$ at node $n$ is 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. Alternatively, one could insert a statement $v := e$ at the point of interest, where $v$ is some dummy variable that did not occur previously in the program, and $e$ is some expression containing all variables in $V$, re-construct the PDG, and slice with respect to the newly added statement. Having dealt with this issue, some conclusions regarding the accuracy of static slicing methods can now be stated.

*Basic algorithms.* For *intra*procedural static slicing, the accuracy of methods based on dataflow equations [3] (see Section 3.1.1) information-flow relations [7] (see Section 3.1.2), and PDGs [4] (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 [3] 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 [7, 36, 61, 67,63] determine for each output parameter precisely which 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 [7] does not compute truly interprocedural slices because only the main program is being sliced. Moreover, 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 [61, 36, 67, 63] compute accurate interprocedural static slices up to the assumption of path feasibility, and are capable of handling recursive programs (see Sections 3.2.2 and 3.2.3). Ernst [50] and Jackson and Rollins [32] also present a solution for interprocedural static slicing that is accurate up to the assumption of path feasibility, but do not present a proof of correctness.

Binkley extended the Horwitz–Reps–Binkley algorithm [36] in two respects: a solution for interprocedural static slicing in the presence of parameter aliasing [64], and a solution for obtaining executable interprocedural static slices [68].

*Unstructured control flow.* Lyle's method for computing static slices in the presence of unstructured control flow is very conservative (see Section 3.3.1). Agrawal [47] has shown that the solutions proposed by Gallagher and Lyle [70, 27] and by Jiang *et al.* are incorrect [59]. Precise solutions for static slicing in the presence of unstructured control flow have been proposed by Ball and Horwitz [45, 46], Choi and Ferrante [48], and Agrawal [47] (see Section 3.3.2). It is our conjecture that these three approaches are equally accurate.

*Composite variables and pointers.* A number of solutions for slicing in the presence of composite variables and pointers were discussed in Section 3.4. Lyle [56] presented a very conservative algorithm for static slicing in the presence of arrays. The approach by Jiang *et al.* [59] for slicing in the presence of arrays and pointers was shown to be incorrect. Lyle and Binkley [74] present an approach for computing highly accurate slices in the presence of pointers, but only consider straight-line code. Agrawal *et al.* propose an algorithm for static slicing in the presence of arrays and pointers that is more accurate than Lyle's algorithm [56].

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

## 3.6.3 *Efficiency*

Below, the efficiency of the static slicing methods that were studied earlier will be addressed.

*Basic algorithms.* Weiser's algorithm for *intra*procedural static slicing based on dataflow equations [3] can determine a slice in $O(v \times (n+e))$ time[16], 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 [7] report that the $\mu_S$ relation for a statement $S$ can be computed in $O(v^2 \times n)$. From $\mu_S$, the slices for all variables at $S$ can be obtained in constant time.

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 unstructured control flow, the control dependences of a single-procedure program can be computed in $O(e)$ in practice [80, 81]. 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. [67]). 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, i.e. $E = O(V^2)$. We are not aware of any slicing algorithms that use more efficient program representations such as the SSA form [82]. However, Yang *et al.* [83] use Program Representation Graphs as a basis for a program integration algorithm that accommodates semantics-preserving transformations. This algorithm is based on techniques similar to slicing.

*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 $p$.

Weiser does not state an estimate of the complexity of his interprocedural slicing algorithm [3]. However, one can observe that for an initial criterion $C$, the set of criteria in $(\text{Up} \cup \text{Down})^*(C)$ contains at most $O(\mathit{Visible} \times \mathit{Sites}_p)$ criteria in each procedure $p$. An intraprocedural slice of procedure $p$ takes time $O(\mathit{Visible} \times (N_p + E_p))$. Furthermore, computation of interprocedural summary information can be done in $O(\mathit{Globals} \times \mathit{TotalSites})$ time [84]. Therefore, the following

[16] Weiser [3] states a bound of $O(n \times e \times \log(e))$; this is a bound on the number of 'bit-vector' steps performed, where the length of each bit-vector is $O(v)$. However, present-day techniques permit computation of the same information in $O(v \times (n+e))$ time.

expression constitutes an upper bound for the time required to slice the entire program:

$$O(Globals \times TotalSites + Visible^2 \times \Sigma_p(Sites_p \times (N_p + E_p)))$$

The approach by Bergeretti and Carré requires that, in the worst case, the $\mu$ relation is computed for each procedure. Each call site is replaced by at most visible assignments. Therefore, the cost of slicing a procedure $p$ is $O(Visible^2 \times (n + Visible \times Sites_p))$, and the total cost of computing a slice of the main program is:

$$O(Visible^2 \times \Sigma_p(n + Visible \times Sites_p))$$

As was discussed in Section 3.2.1, the approach by Hwang *et al.* 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:

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

where *Sites* is the maximum number of call sites in any procedure, and $E^{\mathrm{PDG}}$ is the maximum number of control and data dependence edges in any procedure dependence graph (for details, see [36, 63]). The Reps–Horwitz–Sagiv–Rosay approach for computing summary edges requires

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

time [63]. 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, $V$ and $E$ are the number of vertices and edges in the SDG, respectively.

Binkley does not state a cost estimate of his algorithm for interprocedural slicing in the presence of parameter aliasing [64]. The cost of his 'extension' for deriving executable interprocedural slices [68] from 'non-executable' interprocedural slices is linear in the size of the SDG.

Jackson and Rollins [32], who use an adaptation of Ernst's algorithm for determining summary dependences (see Section 3.2.3) claim a bound of $O(v \times n^2)$, where $v$ denotes the number of variables, and $n$ the number of ports in the dependence graph. Observe that each port is effectively a pair $(variable, statement)$. In the approach by Jackson and Rollins, extraction of a slice is done in a single traversal of the dependence graph, which requires $O(V + E)$ time, where $V$ and $E$ denote the number of vertices (i.e. ports) and edges in the slice.

*Unstructured control flow.* Lyle [56] presented a conservative solution for dealing with unstructured control flow. His algorithm is a slightly modified version of Weiser's algorithm for *structured* control flow [3], which requires $O(v \times (n + e))$ time.

No complexity estimates are stated in [47, 46, 48]. However, the differences between these

**Table 3.** Orthogonal dimensions of static slicing

| *Basis of Algorithm* | *Procedures* | *Unstructured control flow* | *Composite variables* | *Concurrency* |
|---|---|---|---|---|
| Dataflow equations | Weiser [3,55] | Lyle [56] | Lyle [56] | — |
| Inf.-flow relations | Bergeretti, Carré [7] | — | — | — |
| PDG-based | Horwitz *et al.* [36] | Ball, Horwitz [45, 46] | Agrawal *et al.* [73][a] | Cheng [76] |
| | Lakhotia [38] | Choi, Ferrante [48] | | |
| | Reps *et al.* [67, 63] | Agrawal [47] | | |
| | Binkley [68] | | | |

[a] Algorithms for computing conservative approximations of data dependences in the presence of aliasing can be used. See Section 3.4.

algorithms and the 'standard' PDG-based slicing algorithm are only minor: Ball and Horwitz [46] and Choi and Ferrante [48] use a slightly different control dependence subgraph in conjunction with the data dependence subgraph, and Agrawal [47] uses the standard PDG 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 discussed above.

*Composite variables and pointers.* Lyle's approach for slicing in the presence of arrays [56] 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.* [73] 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.

Lyle and Binkley do not state a cost estimate for their approach for slicing in the presence of pointers [74].

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. [85, 86]).

*Concurrency.* Cheng [76] does not state any complexity estimate for determining selection, synchronization, and communication dependence. 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.

### 3.6.4 *Combining static slicing algorithms*

Table 3 highlights 'orthogonal' dimensions of static slicing: dealing with procedures, unstructured control flow, non-scalar variables, and concurrency. 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).

| | | |
|---|---|---|
| $1^1$ | `read(n)` | |
| $2^2$ | `i := 1` | |
| $3^3$ | `i <= n` | `/* (1 <= 2) /*` |
| $4^4$ | `(i **mod** 2 = 0)` | `/* (1 **mod** 2 = 1) /*` |
| $6^5$ | `x := 18` | |
| $7^6$ | `i := i + 1` | |
| $3^7$ | `i <= n` | `/* (2 <= 2) /*` |
| $4^8$ | `(i **mod** 2 = 0)` | `/* (2 **mod** 2 = 0) /*` |
| $5^9$ | `x := 17` | |
| $7^{10}$ | `i := i + 1` | |
| $3^{11}$ | `i <= n` | `/* (3 <= 2) /*` |
| $8^{12}$ | `write(x)` | |

**(a)**

$$\begin{aligned}
\mathrm{DU} &= \{\ (1^1,3^3),\ (1^1,3^7),\ (1^1,3^{11}),\ (2^2,3^3),\ (2^2,4^4),\ (2^2,7^6),\\
&\qquad (7^6,3^7),\ (7^6,4^8),\ (7^6,7^{10}),\ (5^9,8^{12}),\ (7^{10},3^{11})\ \}\\
\mathrm{TC} &= \{\ (3^3,4^4),\ (3^3,6^5),\ (3^3,7^6),\ (4^4,6^5),\ (3^7,4^8),\ (3^7,5^9),\\
&\qquad (3^7,7^{10}),\ (4^8,5^9)\ \}\\
\mathrm{IR} &= \{\ (3^3,3^7),\ (3^3,3^{11}),\ (3^7,3^3),\ (3^7,3^{11}),\ (3^{11},3^3),\ (3^{11},3^7),\\
&\qquad (4^4,4^8),\ (4^8,4^4),\ (7^6,7^{10}),\ (7^{10},7^6)\ \}
\end{aligned}$$

**(b)**

**Fig. 16.** (a) Trajectory for the example program of Fig. 2 (a) for input n = 2. (b) Dynamic flow concepts for this trajectory

## 4. *Methods for dynamic slicing*

### 4.1 *Basic algorithms*

In this section, we will study basic algorithms for dynamic slicing of structured, single-procedure programs with scalar variables.

#### 4.1.1 *Dynamic flow concepts*

Korel and Laski [9, 37] describe how dynamic slices can be computed. 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, Fig. 16 shows the trajectory for the program of Fig. 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[17]. Korel and Laski define a *dynamic slice* with respect to a criterion $(x, I^q, V)$ as an *executable* program $S$ that is obtained from a program $P$ by removing zero or more statements. Three restrictions are imposed on $S$. First, when executed with input $x$, the trajectory of $S$ is identical to the trajectory of $P$ from which all statement instances are removed that correspond to statements that do not occur in $S$. Second, identical values are computed by the program and its slice for all variables in $V$ at the statement occurrence specified in the criterion.

[17] 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 $x$.

| | |
|---|---|
| $1^1$ | `read(n)` |
| $2^2$ | `i := 1` |
| $3^3$ | `i <= n` |
| $4^4$ | `(i mod 2 = 0)` |
| $6^5$ | `x := 18` |
| $7^6$ | `i := i + 1` |
| $3^7$ | `i <= n` |
| $8^8$ | `write(x)` |

**(a)**

$$\begin{aligned} DU &= \{\ (1^1,3^3),\ (1^1,3^7),\ (2^2,3^3),\ (2^2,4^4),\ (2^2,7^6),\ (6^5,8^8),\ (7^6,3^7)\ \} \\ TC &= \{\ (3^3,4^4),\ (3^3,6^5),\ (3^3,7^6),\ (4^4,6^5)\ \} \\ IR &= \{\ (3^3,3^7),\ (3^7,3^3)\ \} \end{aligned}$$

**(b)**

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

**(c)**

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

end;
write(x)
```

**(d)**

**Fig. 17.** (a) Trajectory of the example program of Fig. 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 with respect to the same criterion obtained by ignoring the effect of the IR relation

Third, it is required that statement $I$ corresponding to statement instance $I^q$ specified in the slicing criterion occurs in $S$. Korel and Laski observe that their notion of a dynamic slice has the property that 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* that 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. 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 16 (b) shows the dynamic flow concepts for the trajectory of Fig. 16 (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 before statement instance $I^q$, as well as the test actions in the trajectory on which $I^q$ is control dependent. Approximation $S^{i+1}$ is

```
(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)      z := x;
(8)      i := i + 1
       end;
(9)    write(z)
              (a)
```

```
1¹    read(n)
2²    i := 1
3³    i <= n
4⁴    (i mod 2 = 0)
6⁵    x := 18
7⁶    z := x
8⁷    i := i + 1
3⁸    i <= n
4⁹    (i mod 2 = 0)
5¹⁰    x := 17
7¹¹    z := x
8¹²    i := i + 1
3¹³    i <= n
9¹⁴    write(z)
          (b)
```

**Fig. 18.** (a) Example program. (b) Trajectory for input n = 2

defined as follows:

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

where $A^{i+1}$ is defined as follows:

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

where $q$ is the 'label' of the statement occurrence specified in the slicing criterion. 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.

As an example, the dynamic slice for the trajectory of Fig. 16 and the criterion $(n = 2, 8^{12}, \{x\})$ is computed. 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 Fig. 2 (b).

The role of the IR relation calls for some clarification. Consider the trajectory of the example program of Fig. 2 (a) for input n = 1, shown in Fig. 17 (a). The dynamic flow concepts for this trajectory, and the slice with respect to criterion $(n = 1, 8^8, \{x\})$ are shown in Fig. 17 (b) and (c), respectively. Note that the slice thus obtained is a terminating program: once statement instance $3^3$ is included in the slice, the IR relation will cause the inclusion of $3^7$, which in turn would cause the inclusion of $7^6$ via the DU relation. However, computing the slice without taking the IR relation into account would yield the non-terminating program of Fig. 17 (d). 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

```
(1)  read(n);
(2)  x := 0;
(3)  i := x;
(4)  while (i < n) do
     begin
(5)    x := 1;
(6)    y := 10/x;
(7)    i := i + 1
     end
```

(a)

| | |
|---|---|
| $1^1$ | `read(n)` |
| $2^2$ | `x := 0` |
| $3^3$ | `i := x` |
| $4^4$ | `i < n` |
| $5^5$ | `x := 1` |
| $6^6$ | `y := 10/x` |
| $7^7$ | `i := i + 1` |
| $4^8$ | `i < n` |
| $5^9$ | `x := 1` |
| $6^{10}$ | `y := 10/x` |
| $7^{11}$ | `i := i + 1` |
| $4^{12}$ | `i < n` |

(b)

```
read(n)
x := 0;
i := x;
while (i < n) do
begin
    ;
  y := 10/x;
  i := i + 1
end
```

(c)

**Fig. 19.** (a) Example program. (b) Trajectory for input n = 2. (c) Dynamic slice w.r.t. criterion (n = 2, $6^{10}$,{i}), which produces a run-time error when executed for the same input

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, Fig. 18 (a) shows a slightly modified version of the program of Fig. 2 (a). Figure 18 (b) shows the trajectory for this program. From this trajectory, it follows that $(7^6, 7^{11}) \in \text{IR}$, $(6^5, 7^6) \in \text{DU}$, and $(5^{10}, 7^{11}) \in \text{DU}$. Therefore, both statements (5) and (6) will be included in the slice w.r.t. criterion $(n = 2, 9^{14}, \{z\})$, although statement (6) is neither needed to compute the final value of z nor to preserve termination.

We conjecture that restricting the IR relation to statement instances that correspond to control predicates in the program would yield smaller slices. Alternatively, it would be interesting to investigate if using a dynamic variation of Podgurski and Clarke's notion of weak control dependence [77] could be used instead of the IR relation.

We conclude the discussion of the Korel–Laski algorithm by reporting a minor problem[18]. Consider the example program of Fig. 19 (a). In Fig. 19 (b) the trajectory of that program for input n = 2 is shown, and Fig. 19 (c) shows the dynamic slice of this program with respect to criterion (n = 2, $6^{10}$, {i}). Observe that the statement x := 1 is omitted in the slice, so that the slice will suffer from a division-by-zero error when executed for input n = 2. This problem is due to the fact that the slicing criterion $I^q$ is added to the *final* approximation $S_C$ of the slice, without necessarily adding all statements it is dependent upon. It seems however, that this problem can be resolved by simply initializing $S_0$ with the slicing criterion $I^q$, rather than the last definitions of the variables in $V$.

### 4.1.2 *Dynamic dependence relations*

Gopal [49] describes an approach where *dynamic dependence relations* are used to compute dynamic slices. He introduces dynamic versions of Bergeretti and Carré's information-flow relations[19]

[18] This quirk was originally reported by one of the anonymous referees of this paper.

[19] 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 $\bar{\lambda}_S$, $\bar{\mu}_S$, and $\bar{\rho}_S$ instead.

| | | |
|---|---|---|
| $\bar{\lambda}_\epsilon = \emptyset$ | $\bar{\lambda}_{S_1;S_2} = \bar{\lambda}_{S_1} \cup \bar{\rho}_{S_1} \cdot \bar{\lambda}_{S_2}$ | $\bar{\lambda}_{\nu:=e} = \text{VARS}(e) \times \{e\}$ |
| $\bar{\mu}_\epsilon = \emptyset$ | $\bar{\mu}_{S_1;S_2} = \bar{\mu}_{S_1} \cdot \bar{\rho}_{S_2} \cup \bar{\mu}_{S_2}$ | $\bar{\mu}_{\nu:=e} = \{(e, \nu)\}$ |
| $\bar{\rho}_\epsilon = \text{ID}$ | $\bar{\rho}_{S_1;S_2} = \bar{\rho}_{S_1} \cdot \bar{\rho}_{S_2}$ | $\bar{\rho}_{\nu:=e} = (\text{VARS}(e) \times \{\nu\}) \cup (\text{ID} - (\nu, \nu))$ |

$$\bar{\lambda}_{\text{if } e \textbf{ then } S_1 \textbf{ else } S_2} \begin{cases} (\text{VARS}(e) \times \{e\}) \cup \bar{\lambda}_{S_1} & \text{if } e \text{ evaluates to true} \\ (\text{VARS}(e) \times \{e\}) \cup \bar{\lambda}_{S_2} & \text{if } e \text{ evaluates to false} \end{cases}$$

$$\bar{\mu}_{\text{if } e \textbf{ then } S_1 \textbf{ else } S_2} \begin{cases} (\{e\} \times \text{DEFS}(S_1)) \cup \bar{\mu}_{S_1} & \text{if } e \text{ evaluates to true} \\ (\{e\} \times \text{DEFS}(S_2)) \cup \bar{\mu}_{S_2} & \text{if } e \text{ evaluates to false} \end{cases}$$

$$\bar{\rho}_{\text{if } e \textbf{ then } S_1 \textbf{ else } S_2} \begin{cases} (\text{VARS}(e) \times \text{DEFS}(S_1)) \cup \bar{\rho}_{S_1} & \text{if } e \text{ evaluates to true} \\ (\text{VARS}(e) \times \text{DEFS}(S_2)) \cup \bar{\rho}_{S_2} & \text{if } e \text{ evaluates to false} \end{cases}$$

**Fig. 20.** Definition of dynamic dependence relations

$\lambda_S$, $\mu_S$, and $\rho_S$ (see Section 3.1.2). The $\bar{\lambda}_S$ relation contains all pairs $(v, e)$ such that statement $e$ depends on the input value of $v$ when program $S$ is executed. Relation $\bar{\mu}_S$ 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 $\bar{\rho}_S$ 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 [7], the dependence relation for a **while** statement (omitted here) is expressed in terms of dependence relations for nested conditionals with equivalent behaviour. However, whereas 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 20 summarizes Gopal's dynamic dependence relations. Here, DEFS($S$) denotes the set of variables that is actually modified by executing statement $S$. Note that this definition of DEFS is 'dynamic' in the sense that it takes into account which branch of an **if** statement is executed. Using these relations, a dynamic slice w.r.t. the final value of a variable $v$ is defined as:

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

Figure 21 (a) shows the information-flow relation $\bar{\mu}$ for the (entire) program of Fig. 2 (a)[20]. From

[20] 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.

| *Expression number*[a] | *Affected variables* |
|---|---|
| 1 | {i,n,x} |
| 2 | {i,x} |
| 3 | {i,x} |
| 4 | {i,x} |
| 5 | {x} |
| 6 | $\emptyset$ |
| 7 | {i,x} |
| 8 | $\emptyset$ |

[a] Expressions are indicated by the line numbers in Fig. 2 (a)

**Fig. 21.** (a) The $\bar{\mu}$ relation for the example program of Fig. 2 (a) and input n = 2

this relation it follows that the set of expressions that affect the value of x at the end of the program for input n = 2 are {1, 2, 3, 4, 5, 7}. The corresponding dynamic slice is almost identical to the one shown in Fig. 1 (b), the only difference being the fact that Gopal's algorithm excludes the final statement write(x) on line 8.

For certain cases, Gopal's algorithm may compute a non-terminating slice of a terminating program. Figure 22 (a) shows the slice for the program of Fig. 2 (a) 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, Fig. 22 (b) shows the slice with respect to the final value of z for the example program of Fig. 18 (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 Gopal's slice.

### 4.1.3 *Dependence graphs*

Miller and Choi [11] were the first to introduce a dynamic variation of the PDG, called the *dynamic program dependence graph*. These graphs are used by their parallel program debugger to perform *flowback analysis* [10] and are constructed incrementally, on demand. Prior to execution, a (variation of a) static PDG is constructed, and the object code of the program is augmented with code that 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.

Agrawal and Horgan [35] 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[21], and marks the *vertices* that are executed for a given test set. A dynamic slice is determined by computing a static slice in the subgraph of the PDG that is induced by the marked

[21] The dependence graphs of [35] 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 this paper have an entry vertex.

```
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)

**Fig. 22.** (a) Non-terminating slice computed for the example program of Fig. 2 (a) with respect to the final value of x, for input n = 1. **(b)** Slice for the example program of Fig. 18 (a) with respect to the final value of x, for input n = 2

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, Fig. 23 (a) shows the PDG of the example program of Fig. 2 (a). Suppose that 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 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 23 (b) shows the PDG of the example program of Fig. 18 (a). For input n = 2, all dependence edges will be marked, causing the slice to consist of the entire program. It is shown in [35] 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 results that are identical[22] to those of 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 require $O(N)$ space, where $N$ denotes the number of executed statements.

Agrawal and Horgan's second approach computes overly large slices because it does not account

[22] Actually, there is a minor difference, which has to do with the fact that the slices computed by the Korel–Laski algorithm may give rise to run-time errors when executed. This was discussed in the last paragraph of Section 4.1.1.

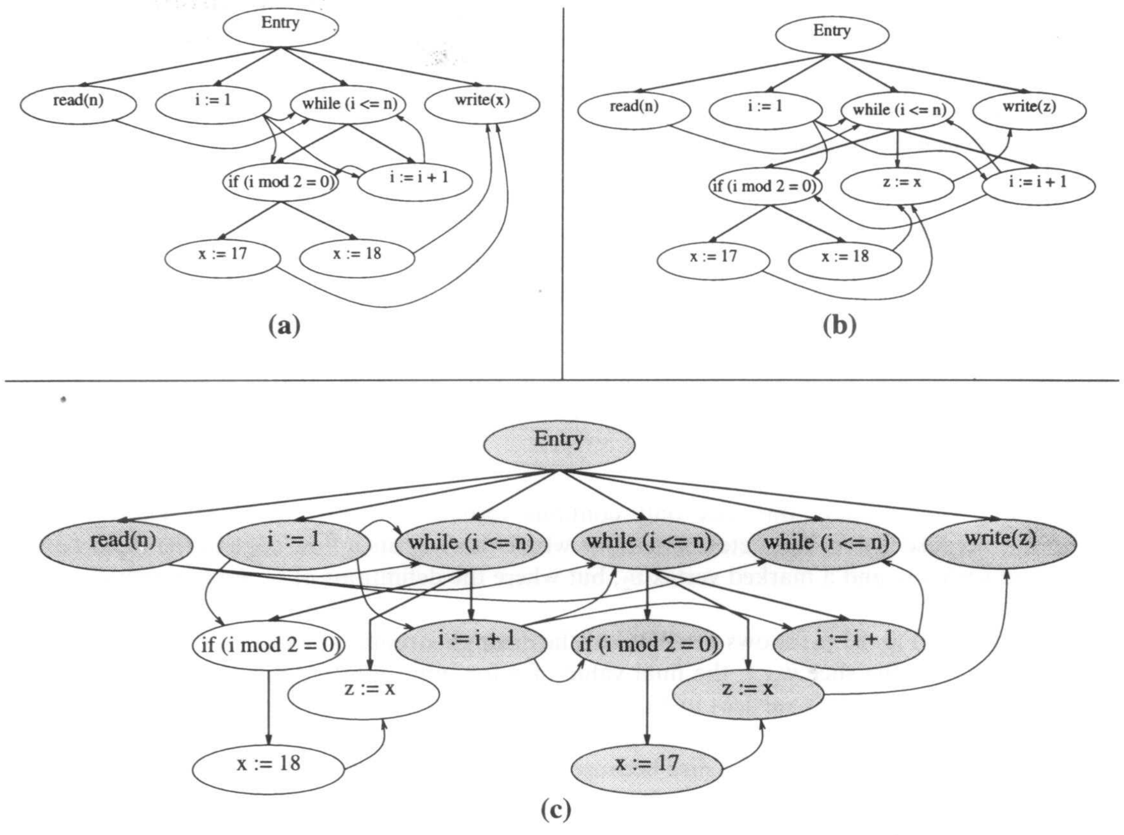


**Fig. 23.** (a) PDG of the program of Fig. 2 (a). (b) PDG of the program of Fig. 18 (a). (c) DDG of the program of Fig. 18 (a)

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 23 (c) shows the DDG for the example program of Fig. 18 (a). The slicing criterion corresponds to the vertex labelled `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 labelled `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. 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 24 shows a program $\mathtt{Q}^n$ that has

```
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.
```

**Fig. 24.** Program Q$^n$ with $O(2^n)$ different dynamic slices

$O(2^n)$ dynamic slices. The program reads a number of values in variables $x_i (1 \le i \le n)$, 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 $2^n$ different 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 Fig. 23 (c), the vertices labelled `i := i + 1` and the rightmost two vertices labelled `i <= n` have the same transitive dependences; these vertices depend on statements 1, 2, 3, and 8 of the program of Fig. 18 (a). Hence, the RDDG for this program (given input n = 2) is obtained by merging these four DDG vertices into one vertex.

Agrawal and Horgan [35] present an algorithm for the construction of an RDDG without having to 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 Procedures

Agrawal *et al.* [73] consider dynamic slicing of procedures with call-by-value, call-by-reference, call-by-result, and call-by-value-result parameter-passing. A key property of their method is that dynamic data dependences are defined in terms of definitions and uses of *memory locations*; this has the advantage that global variables do not require special treatment, and that no alias analysis is necessary. Agrawal *et al.* describe how each parameter passing mechanism can be modelled by a set of mock assignments that is inserted before and/or after each procedure call. In the subsequent discussion, it is assumed that a procedure $P$ with formal parameters $f_1, \cdots, f_n$ is called with actual parameters $a_1, \cdots, a_n$. *Call-by-value* parameter-passing can be modelled by a sequence of assignments $f_1 := a_1; \cdots; f_n := a_n$ that 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, whereas the Def sets for the formal parameters $f_i$ are determined *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 *et al.* [87, 88]. 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 affects 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[23]. 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: this may result in inaccurate slices in the presence of conditionals. In the second approach, all dependences are determined at run-time. While this results in more 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

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

inside basic blocks statically, and the inter-block dependences dynamically. In all approaches control dependences[24] 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.

Kamkar [14] adapts the interprocedural slicing method by Kamkar *et al.* [87, 88] 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 *et al.* [12] discuss an approach for interprocedural flowback analysis. 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 either to 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 Composite data types and pointers

### 4.3.1 *Dynamic flow concepts*

Korel and Laski [37] 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 *et al.* [73] present a dependence graph based algorithm for dynamic slicing in the presence of composite data types and pointers. Their solution consists of expressing DEF and USE sets in terms of *actual memory locations* provided by the compiler. The algorithm of [73] is similar to that for *static* slicing in the presence of composite data types 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 *et al.* [12] extend the flowback analysis method by Miller and Choi [11] (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 that are either deleted, or changed into genuine data dependences at run-time. Pointer accesses are resolved at run-time, by recording all uses of pointers in the log file.

[24] Kamkar *et al.* use a notion of *termination-preserving* control dependence that is similar to Podgurski and Clarke's *weak* control dependence [77].

## 4.4 Concurrency

### 4.4.1 *Dynamic flow concepts*

Korel and Ferguson [89] extend the dynamic slicing method of Korel and Laski [9, 37] to distributed programs with Ada-type rendezvous communication (see, e.g. [90]). For a distributed program, the execution history is formalized as a *distributed program path* that, for each task, comprises: (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 behaviour 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 Korel–Ferguson method for computing slices of distributed programs of [89] is basically a generalization of the Korel–Laski method [9, 37], 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 that uses a separate slicing process for each task.

### 4.4.2 *Dependence graphs*

Duesterwald *et al.* [91] present a dependence graph based algorithm for computing dynamic slices of distributed programs. They introduce a Distributed Dependence Graph (DDG)[25] 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 synchronous 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. Slices are computed in the

[25] 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.

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 Fig. 18 with input n = 2 will be the entire program.

Cheng [76] presents an alternative dependence graph based algorithm for computing dynamic slices of distributed and concurrent programs. 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 [35]: 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.

Choi *et al.* [11, 12] 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 that contains synchronization vertices for synchronization operations (such as *P* and *V* on a semaphore) and synchronization edges that 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 Comparison

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 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 concurrency 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. The work by Field *et al.* [17, 18, 92, 79] mentioned in Table 4 relies on substantially different techniques than those used for the dynamic slicing algorithms discussed previously, and will therefore be studied separately in Section 6.

**Table 4.** Overview of dynamic slicing methods

| *Reference* | *Computation method*[a] | *Executable* | *Interprocedural solution* | *Data types*[b] | *Concurrency* |
|---|---|---|---|---|---|
| Korel, Laski [9, 37] | D | yes | no | S, A, P | no |
| Korel, Ferguson [89] | D | yes | no | S, A | yes |
| Gopal [49] | I | no | no | S | no |
| Agrawal, Horgan [35] | G | no | no | S | no |
| Agrawal *et al.* [93, 73] | G | no | yes | S, A, P | no |
| Kamkar *et al.* [87, 88] | G | no | yes | S | no |
| Duesterwald *et al.* [91] | G | yes | no | S, A, P | yes |
| Cheng [76] | G | no | no | S | yes |
| Choi *et al.* [11, 12] | G | no | yes | S, A, P | yes |
| Field *et al.* [17, 18, 92, 79] | R | yes | no | S, A, P | no |

[a] D = dynamic flow concepts, I = dynamic dependence relations, G = reachability in a dependence graph. R = dependence tracking in term graph rewriting systems (see Section 6).
[b] S = scalar variables, A = arrays/records, P = pointers.

Unlike in the static case, there exists a significant difference between methods that compute executable slices [9, 37, 91, 89], and approaches that compute slices merely consisting of sets of statements [35, 73, 49]. The latter type of slice may not be executable due to the absence of assignments for incrementing loop counters[26]. For convenience, we will henceforth refer to such slices as 'non-executable' slices. As was 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 [15], there is very little semantic justification for any of the methods for computing 'non-executable' slices. The algorithms of [11, 35, 87, 76, 88] 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 [9, 37] (see Section 4.1.1) are larger than those computed by the algorithms of Agrawal and Horgan [35] (see Section 4.1.3) and Gopal [49] (see Section 4.1.2). This is due to Korel and Laski's constraint that their slices should be executable.

*Procedures.* Dependence graph based algorithms for interprocedural dynamic slicing were proposed by Agrawal *et al.* [73], and by Kamkar *et al.* [87, 88] (see Section 4.2). It is unclear if one of these algorithms produces more accurate slices than the other.

*Composite variables and pointers.* Korel and Laski [37] (see Section 4.1.1), and Agrawal *et al.* (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.

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

*Concurrency*. Korel *et al.* [89] (see Section 4.4.1) and Duesterwald *et al.* [91] (see Section 4.4.2) compute executable slices, but deal with nondeterminism in a different way. The former approach is based on a mechanism for replaying rendezvous in the slice in the same relative order as they appeared in the original program by analysing a previously stored log file. The latter approach replaces nondeterministic communication statements that occur in the program by deterministic communication statements in the slice, so that the slice can be re-executed yielding similar results. Cheng [76] and Choi *et al.* [11, 12] (see Section 4.4.2) do not address this problem because the slices they compute are not necessarily executable. The dynamic slicing methods by Cheng and Duesterwald *et al.* are inaccurate because they are based on 'static' dependence graphs in which no distinction is made between the different occurrences of a statement in the execution history (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 [87, 88]) $N$. 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. [9, 37, 89]) 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 (sometimes even constant) time. Whenever time requirements are discussed below, the time spent during execution for constructing histories or dependence graphs will be ignored. Space requirements will always be discussed in detail.

*Basic algorithms*. Korel and Laski's solution [9, 37] (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 [49] (see Section 4.1.2) requires $O(N)$ space to store the execution history and $O(n \times v)$ space to store the $\bar{\mu}_S$ relation. The time required to compute the $\bar{\mu}_S$ 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 was 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, slices can be extracted from it in $O(1)$.

*Procedures*. The interprocedural dynamic slicing method proposed by Kamkar *et al.* [87, 88] (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 *et al.* [73] are essentially the same as those of the Agrawal–Horgan basic slicing method discussed above.

*Composite variables and pointers*. The algorithms by Korel and Laski [37] (see Section 4.3.1) and Agrawal *et al.* [73] (see Section 4.3.2) for slicing in the presence of composite variables and pointers

Table 5. Orthogonal dimensions of dynamic slicing

| *Basis of alogrithm* | *Procedures* | *Composite variables* | *Concurrency* |
|---|---|---|---|
| Dyn. flow concepts | — | Korel, Laski [9, 37] | Korel, Ferguson [89] |
| Dyn. dependence relations | Gopal [49] | — | — |
| Dependence graphs | Agrawal *et al.* [73]<br>Kamkar *et al.* [87, 88] | Agrawal *et al.* [73] | Duesterwald *et al.* [91]<br>Cheng [76]<br>Choi *et al.* [11, 12] |

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 are used to determine data dependences, do not affect the worst-case behaviour of the algorithms. Therefore, we expect the time and space requirements to be the same as in the scalar variable case.

*Concurrency.* The algorithms by Cheng [76] and Duesterwald *et al.* [91] 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 by Duesterwald *et al.* [91] 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' dimensions 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 [37], and concurrency [89] 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.* [73], Kamkar *et al.* [87, 88] and Choi *et al.* [11, 12] use dynamic dependence graphs with distinct vertices for different occurrence of statements in the execution history. In contrast, Duesterwald *et al.* [91] and Cheng [76] use variations of static PDGs.
- The dynamic slicing approach of Agrawal *et al.* [73] 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.* [87, 88] 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.

## 5. Applications of program slicing

### 5.1 *Debugging and program analysis*

Debugging can be a difficult task when one is confronted with a large program, and few 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 [56]. 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. In this case, it is *likely* that the error occurs in one of the statements in the slice. However, it need not *always* be the case that the error occurs in the slice, as an error may consist of a statement that is missing inadvertently. However, in situations like this it is probable that more, or different statements show up in the slice than one would expect.

Forward slices are also useful for debugging. A forward slice w.r.t. a statement $s$ can show how a value computed at $s$ is being used subsequently, and can help the programmer ensure that $s$ establishes the invariants assumed by the later statements. For example, this can be useful in catching off-by-one errors. Another purpose of forward slicing is to inspect the parts of a program that may be affected by a proposed modification, to check that there are no unforeseen effects on the program's behaviour.

Lyle and Weiser [19] 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 that 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* by Agrawal [93]). The authors claim that program dicing still produces useful results when these assumptions are relaxed.

Bergeretti and Carré [7] explain how static slicing methods can detect 'dead' code, i.e. statements that cannot affect any output of the program. Often, such statements are not executable due to the presence of a bug. Static slicing can also be employed to determine uses of uninitialized variables, another symptom of an error in the program. However, there exist previous techniques for detection of dead code and uses of uninitialized variables [65, 86] that do not rely on slicing.

In debugging, one is often interested in a specific execution of a program that exhibits anomalous behaviour. *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 dissertation [93] contains a detailed discussion about how static and dynamic slicing can be utilized for semi-automated debugging of programs [35, 73]. 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. 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 Lyle and Weiser [19]. Obviously, these operations for combining slices may produce false leads in the presence of multiple bugs or coincidental correctness. Agrawal *et al.* [20] discuss the implementation of a debugging tool that is based on ideas in previous papers by the same authors [93, 73, 35].

Pan and Spafford [22, 23] present a number of heuristics for fault localization. These heuristics describe how dynamic slices (variations on the type proposed by Agrawal *et al.* [35]) can be used for selecting a set of suspicious statements that is likely to contain a bug. The approach of Pan and Spafford consists of two phases. First, the program is executed for an extensive number of test cases, and each test case is classified as being *error-revealing* or *non-error-revealing*, depending on whether or not its reveals the presence of a bug. The second step consists of the actual *heuristic rules* for combining the information contained in dynamic slices for these test cases in various ways. As an example, one might think of displaying the set of statements that occur in *every* dynamic slice for an error-revealing test-case—such statements are likely to contain the bug. Other heuristics depend on the *inclusion frequency* or the *influence frequency* of statements in dynamic slices. The former denotes the number of slices in which a particular statement occurs, whereas the latter notion indicates the number of times that a statement in a particular dynamic slice is 'referred to' in terms of data dependence and control dependence. For example, one of the heuristics given by Pan and Spafford consists of selecting the statements with 'high' influence frequency in a slice for a selected error-revealing test case. Note that this requires a *threshold* to be specified by the user that determines the boundary between 'high' and 'low' frequencies. It is argued that this boundary can be shifted interactively, thereby gradually increasing the number of statements under consideration.

Choi *et al.* [12] describe the design and efficient implementation of a debugger for parallel programs that 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 [21] and dynamic [87, 14] slicing for algorithmic debugging [94, 95]. An algorithmic debugger partially automates the task of localizing a bug by comparing the *intended* program behaviour with the *actual* program behaviour. The intended behaviour 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.

## 5.2 *Program differencing and program integration*

Program *differencing* [25] is the task of analysing 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 behaviour 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 behaviours. The program integration algorithm of Horwitz *et al.* [26] discussed below, compares slices in order to detect equivalent behaviours. However, an alternative partitioning technique by Yang *et al.* [83, 25], which is not based on comparing slices but on comparing smaller units of code, produces more accurate results because semantics-preserving transformations (e.g. copy propagation) can be accommodated.

Horwitz *et al.* [26] use the static slicing algorithm for single-procedure programs by Horwitz *et al.* [36] as a basis for an algorithm that integrates changes in variants of a program. The inputs of their algorithm consist of a program *Base*, and two variants $A$ and $B$ that 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 $G_A$ ($G_B$) that have a different slice in $G_{Base}$[27].
(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 behaviours 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.

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 Horwitz and Reps [43] for comparing two slices in time that is linear in the sum of their sizes could be used. A semantic justification for the single-procedure slicing algorithm of Horwitz *et al.* [36] and the program integration algorithm of Horwitz *et al.* [26] is presented by Reps and Yang [42]. This paper formalizes the relationship between the execution behaviours of programs, slices of those programs, and between variants of a program and the corresponding integrated version.

Reps [96] presents an alternative formulation of the Horwitz–Prins–Reps program integration algorithm that is based on Brouwerian algebras. The algebraic laws that hold in such algebras are used to restate the algorithm and to prove properties such as associativity of consecutive integrations.

Binkley *et al.* [97] generalize the integration algorithm of Horwitz *et al.* [26] to multi-procedure programs. It is shown that such programs cannot be integrated on a per-procedure basis (program behaviour would not be preserved in all cases), and that a straightforward extension using the Horwitz–Reps–Binkley interprocedural slicing algorithm is insufficiently powerful (it reports 'interference' in too many cases). While a complete discussion of the theory that underlies the Binkley–Horwitz–Reps multi-procedure integration algorithm is outside the scope of this paper,

[27] 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 that have a different set of incoming edges in $G_A$ and in $G_{Base}$ [40].

it can be remarked here that the algorithm relies on backward and forward interprocedural slices on the SDG representation of the program.

## 5.3 *Software maintenance*

One of the problems in software maintenance consists of determining whether a change at some place in a program will affect the behaviour of other parts of the program. Gallagher and Lyle [70, 27] 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 behaviour. They present a set of guidelines for the maintainer of a component that, if obeyed, preclude changes in the behaviour 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. Intuitively, a decomposition slice captures part of the behaviour of a program, and its complement captures the behaviour of the rest of the program. A decomposition slice w.r.t. a variable $v$ is defined as the set 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 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 that is not strongly dependent on any other ORD slice is *maximal.* A statement that occurs in more than one ORD slice is *dependent*; otherwise it is *independent.* A variable is *dependent* if it is assigned to some dependent statement; it is *independent* if it is only assigned to 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.

The essential observation by Gallagher and Lyle [27] 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 behaviour.

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.

## 5.4 *Testing*

A program satisfies a 'conventional' *data flow testing* criterion if all def-use pairs occur in a successful test-case. Duesterwald *et al.* [13] propose a more rigorous testing criterion, based on program slicing: 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 of Horwitz *et al.* [36]), dynamic slices are computed using *dynamic dependence graphs* (similar to the DDGs of Agrawal and Horgan [35], but instances of the same vertex are merged, resulting in a slight loss of precision), and *hybrid slices* are computed using dependence graphs that are based on a combination of static and dynamic information. 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.

Kamkar *et al.* [28] 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 by Kamkar *et al.* [87, 88] 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 that was discussed in 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 *et al.* [29] describe an approach to regression testing where slicing techniques are used. Backward and forward static slices serve to determine the program parts affected by the change, and only test cases that 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 by Gupta *et al.* [29] are designed to determine the information necessary for regression testing only (i.e. affected def-use pairs).

Binkley [98] describes an approach for reducing the cost of regression testing of *multi-procedure* programs by (i) reducing the number of tests that must be re-run, and (ii) decreasing the size of the program that they must run on. This is accomplished by determining the set of program points *affected* by the modification, and the set of *preserved* program points (see Section 5.2). The set of affected points is used to construct a smaller and more efficient program that only captures the modified behaviour of the original program; all test-cases that need to be re-run can be applied to this program. The set of preserved points is used to infer which test-cases need not be re-run.

Bates and Horwitz [30] use a variation of the PDG notion of Horwitz *et al.* [26] 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 that did not occur previously as well as any statements that 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 prove that statements in the same class are exercised by the same test cases.

## 5.5 *Tuning compilers*

Larus and Chandra [34] present an approach for tuning of compilers where dynamic slicing is used to detect potential occurrences of redundant common subexpressions. 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) that constructs dynamic slices w.r.t. the current values stored in registers. Larus and Chandra use a variation of the approach by Agrawal and Horgan [35]: a dynamic slice is represented by a 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).

## 5.6 *Other applications*

Weiser [24] describes how slicing can be used to *parallelize* the execution of a sequential program. 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 behaviour 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 behaviour of the original program. Splicing does not rely on a particular slicing technique; any method for computing executable static slices is adequate. Only programs with structured control flow are considered, because Weiser's splicing algorithm depends on the fact that execution behaviour 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 behaviour becomes unsolvable in the presence of irreducible control flow.

Ott and Thuss [99] view a module as a set of processing elements that 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. *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.

Binkley [78] presents a graph rewriting semantics for System Dependence Graphs that is used for performing interprocedural constant propagation. The Horwitz–Reps–Binkley interprocedural slicing algorithm is used to extract slices that may be executed to obtain constant values.

Beck and Eichmann [31] 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.

Jackson and Rollins present a reverse engineering tool called 'Chopshop' in [33] that is based on the techniques of [32] (see Sections 3.1.3 and 3.2.3). This tool provides facilities for visualizing program slices in a graphical manner as diagrams. In addition to 'chopping' (see Section 3.1.3), their tool is capable of 'abstracting' slices by eliminating all non-call-site nodes in a graph and resulting in a graph with only call site vertices and transitive dependence edges between these vertices.

Ning *et al.* [16] 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.

```
read(n);
i := 1;
if (i > 0) then
  n := n + 1
else
  n := n * 2;
write(n)
```

**(a)**

```
read(n);
i := 1;
if (i > 0) then
  n := n + 1
else
           ;
write(n)
```

**(b)**

```
read(n);


  n := n + 1


           ;
write(n)
```

**(c)**

**Fig. 25.** (a) Example program = static slice with respect to statement `write(n)`. (b) More accurate slice obtained by employing constant propagation. (c) Minimal slice

## 6. Recent developments

This section is concerned with recent work on improving the precision of slicing methods, which relies on the removal of two important restrictions characteristic of the slicing algorithms discussed previously:

(1) The fact that a slice consists of a subset of the statements of the original program, sometimes with the additional constraint that a slice must constitute a syntactically valid program.
(2) The fact that slices are computed by tracing data and control dependences.

Both of these 'restrictions' adversely affect the accuracy of the computed slices. Moreover, it is important to realize that these issues are strongly interrelated in the sense that, in many cases, dismissing the former constraint is a prerequisite for being able to dismiss the latter one.

Weiser already observed some problems caused by the first constraint in his dissertation [1, page 6], where he states that 'good source language slicing requires transformations beyond statement deletion'. This remark can easily be understood by considering a situation where a programming language does not allow **if** statements with empty branches, but where a slicing algorithm would exclude all statements in such a branch. Taken to the extreme, such statements can never be removed from a slice because the result would not be a syntactically valid program. Hwang *et al.* [100] discuss a number of related problems and conclude that, in practice, statement deletion alone is an inadequate method for deriving slices.

The second constraint—the fact that slices are to be computed by tracing data and control dependences alone—has to be removed as well, if the singular objective is to compute slices that are as small as possible. To see this, consider the example program of Fig. 25 (a). Here, the static slice with respect to statement `write(n)` as computed by any of the 'conventional' slicing algorithms consists of the entire program[28]. However, if constant propagation [101] or similar optimization techniques could be used in slicing, the resulting slices might be more accurate. In the program of Fig. 25 (a), for example, one can determine that the value of `i` is constant, and that the **else** branch of the conditional is never selected. Therefore, computation of the more accurate slice of Fig. 25 (b) is conceivable. Moreover, if replacement of an entire **if** statement

[28] Some algorithms [3, 7] would omit the `write` statement.

```
read(p);
read(q);
if (p = q) then
  x := 18
else
  x := 17;
if (p <> q) then
  y := x;
else
  y := 2;
write(y)
```

**(a)**

```
read(p);
read(q);
if (p = q) then
  begin
    x := 18;
    y := 2
  end
else
  begin
    x := 17;
    y := x
  end
write(y)
```

**(b)**

```
read(p);
read(q);
if (p = q) then
          ;
else
  x := 17;
if (p <> q) then
  y := x;
else
  y := 2;
write(y)
```

**(c)**

**Fig. 26.** (a) Example program = static slice with respect to the statement `write(y)`. (b) Transformed program. (c) More accurate slice obtained by slicing in the transformed program

by one of the statements in its branches is allowed, one can imagine that the minimal slice of Fig. 25 (c) is determined.

Other compiler optimization techniques[29], symbolic execution, and a variety of semantics-preserving transformations can also be used for obtaining more accurate slices. For example, Fig. 26 (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 Fig. 26 (b). Clearly, a slicing algorithm that can (conceptually) perform this transformation is in principle capable of determining the more accurate slice of Fig. 26 (c).

Approaches that use optimization techniques for obtaining more accurate slices, such as the ones shown in Figs 25 and 26, were presented by Field *et al.* [17, 18, 79], and by Ernst [50]. At the conceptual level, these slicing approaches rely on the following components:

- Translation of the program to a suitable intermediate representation (IR).
- Transformation and optimization of the IR.
- Maintaining a mapping between the source-text, the original IR, and the optimized IR.
- Extraction of slices from the IR.

Field *et al.* [17, 18, 79] use an intermediate representation for imperative programs named PIM [102] as a basis for their slicing approach. Both the translation of a program to its PIM representation, and subsequent optimizations of PIM graphs are defined by an equational logic, which can be implemented by term rewriting [103] or graph rewriting [104]. Correspondences between the source text of a program, its initial PIM graph, and the subsequently derived optimized PIM graph are automatically maintained by a technique called *dynamic dependence tracking* [17, 79]. This technique, which is defined for arbitrary term rewriting systems, keeps track of the way in which new

[29] See, e.g. [86] for a comprehensive overview.

vertices that are dynamically created in a rewriting process are *dependent* upon vertices that were previously present. These source correspondences are stored in PIM graphs as annotations of vertices; in a sense this is similar to the way information is stored in the Reduced Dynamic Dependence Graphs of Agrawal *et al.* [35] (see Section 4.1.3). Extracting a slice with respect to a designated expression involves maintaining a pointer to the PIM-subgraph for that expression, and retrieving the dynamic dependence information stored in that PIM-subgraph. For details as to how this is accomplished, the reader is referred to [18, 79].

Both PIM and dynamic dependence tracking have been implemented using the ASF + SDF Meta-environment, a programming environment generator [105] developed at CWI. Recent experiments have produced promising results. In particular, the (accurate) slices of Figs 25 (b) and 26 (c) have been computed. Recently, Tip [92, 79] has shown that dynamic dependence tracking can also be used to compute accurate dynamic slices from a simple algebraic specification [106] that specifies an interpreter.

Ernst [50] uses the Value Dependence Graph (VDG) [107] as an intermediate representation for his slicing technique. The nodes of a VDG correspond to computations, and the edges represent values that flow between computations. The most prominent characteristics of VDGs are: (i) control flow is represented as data flow; (ii) loops are modelled by recursive function calls; and (iii) all values and computations in a program, including operations on the heap and on I/O streams, are explicitly represented in the VDG. The transformation/optimization of VDGs is discussed in some detail in [107]. Ernst refers to the problem of maintaining a correspondence between the VDG and the source code graph throughout the optimization process, but no details are presented as to how this is accomplished. For the extraction of slices from a VDG, Ernst uses a simple and efficient graph reachability algorithm similar to the one used by Ottenstein and Ottenstein [4].

We are currently unable to provide an in-depth comparison of the approaches by Field *et al.* and by Ernst due to the elaborate optimizations that are involved, and the absence of any information regarding the 'source correspondences' used by Ernst. A few differences between these works are obvious, however:

- The language considered by Ernst is substantially larger than the one studied in the Field–Ramalingam–Tip paper. Ernst has implemented a slicer for the full C language (including recursive procedures—see Section 3.2.3), whereas Field *et al.* do not (yet) address the problems posed by procedures and unstructured control flow.
- The approach by Field *et al.* permits the use of a number of variations of the PIM logic for treating loops, corresponding to different 'degrees of laziness' in the language's semantics. Depending on the selected option, the computed slices will resemble the 'non-executable' slices computed by Agrawal and Horgan [35], or the 'executable' slices computed by Korel and Laski [37]. It is unclear from Ernst's paper if his approach provides the same degree of flexibility.
- Field *et al.* permit slices to be computed given any set of constraints on a program's inputs, and define the corresponding notion of a *constrained* slice, which subsumes the traditional concepts of static and dynamic slices. This is accomplished by rewriting PIM graphs that contain variables (corresponding to unknown values) in combination with PIM-rules that model symbolic execution. Ernst does not discuss a similar capability of his slicer.

```
*(ptr = &a) = ?A;
b = ?B;
x = a;
if (a < 3)
  ptr = &y;
else
  ptr = &x;
if (b < 2)
  x = a;
(*ptr) = 20;
```

**(a)**

```
*(□ = &a) = ?A;
b = □;
x = a;
if (a < 3)
  ptr = &y;
else
  □
if (□ < □)
  x = a;
(*ptr) = □;
```

**(b)**

```
*(□ = &a) = ?A;
b = □;
x = □;
if (a < 3)
  □
else
  ptr = &x;
if (□ < □)
  x = □;
(*ptr) = 20;
```

**(c)**

**Fig. 27.** (a) An example program. (b) Constrained slice with respect to the final value of x given the constraint ?A := 2. (c) Conditional constrained slice with respect to the final value of x given the constraint ?A > 5

- Field *et al.* define slices as a *subcontext* of (i.e. a 'connected' set of function symbols in) a program's AST. Statements or expressions of the program that do not occur in the slice are represented by 'holes' (i.e. missing subterms) in a context. Although this notion of a slice does not constitute an executable program in the traditional sense, the resulting slices *are* executable in the sense that such as slice can be rewritten to a PIM graph containing the same value for the expression specified in the slicing criterion, given the same constraints on the program's inputs.

Figure 27 shows an example program (taken from [18]), and some constrained slices of it obtained using the approach by Field *et al.*[30]. The intuition behind these slices is quite simple: a 'boxed' expression in a slice may be replaced by any other expression without affecting the computation of the value specified in the slicing criterion, given the specified constraints on the program's inputs. Although absurdly contrived, the example illustrates several important points. By not insisting that a slice be a syntactically valid program, distinctions can be made between assignment statements whose R-values are included but whose L-values are excluded and vice versa, as Fig. 27 (b) shows. Observe that it is possible to determine that the values tested in a conditional are irrelevant to the slice, even though the body is relevant. In general, this permits a variety of fine distinctions to be made that traditional slicing algorithms cannot achieve.

## 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, unstructured control

[30] In this figure, expressions that begin with a question mark, e.g. '?A', represent unknown values or inputs. Subterms of the program's AST that do not occur in the slices of Fig. 27 (b) and (c) are replaced by a box.

flow, composite variables/pointers, and concurrency. 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 that capture a part of the program's behaviour. Wherever possible, different solutions to the same problem were compared by applying each algorithm to the same example program. In addition, the possibilities and problems associated with the integration of solutions for 'orthogonal' language features were discussed.

## 7.1 *Static slicing algorithms*

In Section 3.6, algorithms for static slicing were compared and classified. Besides listing the specific slicing problems studied in the literature, we have compared the *accuracy* and, to some extent, the *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 data types and pointers, and concurrency, the accuracy of methods based on dataflow equations [3], information-flow relations [7], and program dependence graphs [4] 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 [3], 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. The solution by Bergeretti and Carré [7] 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 *et al.* [61], Reps *et al.* [36, 67, 63], Jackson and Rollins [32], and Ernst [50]. The Reps–Horwitz–Sagiv–Rosay algorithm for interprocedural static slicing is the most efficient of these algorithms. Binkley studied the issues of determining executable interprocedural slices [68], and of interprocedural static slicing in the presence of parameter aliasing [64].

*Unstructured control flow.* Lyle was the first to present an algorithm for static slicing in the presence of unstructured control flow [56]. The solution he presents is conservative: it may include more **goto** statements than necessary. Agrawal [47] has shown that the solutions proposed by Gallagher and Lyle [70, 27] and by Jiang *et al.* [59] are incorrect. Precise solutions for static slicing in the presence of unstructured control flow have been proposed by Ball and Horwitz [45, 46], Choi and Ferrante [48], and Agrawal [47]. It is not clear how the efficiency of these algorithms compares.

*Composite data types/pointers.* Lyle [56] presented a conservative algorithm for static slicing in the presence of arrays. The algorithm proposed by Jiang *et al.* in [59] is incorrect. Lyle and Binkley [74] presented an algorithm for static slicing in the presence of pointers, but only for straight-line code. Agrawal *et al.* [73] propose a PDG-based algorithm for static slicing in the presence of composite variables and pointers.

*Concurrency.* The only approach for static slicing of concurrent programs was proposed by Cheng [76]. Unfortunately, Cheng has not provided a justification of the correctness of his algorithm.

## 7.2 Dynamic slicing algorithms

Algorithms for dynamic slicing were compared and classified 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' dimensions 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 data types and pointers, and concurrency were proposed by Korel and Laski [9, 37], Agrawal and Horgan [35], and Gopal [49]. The slices determined by the Agrawal–Horgan algorithm and the Gopal algorithm are smaller 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 amount of space proportional to the number of statements that was executed 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 by Choi *et al.* [11, 12]. 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 *et al.* [73], and by Kamkar *et al.* [87, 88]. 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 modelled 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 [73]. 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 Agrawal *et al.* 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.

*Unstructured control flow.* As far as we know, dynamic slicing in the presence of unstructured control flow has not been studied yet. However, it is our conjecture that the solutions for the static case [45–48] may be adapted for dynamic slicing.

*Composite data types/pointers.* Two approaches for dynamic slicing in the presence of composite data types and pointers were proposed, by Korel and Laski [37], and Agrawal *et al.* [73]. The algorithms differ in their computation method: dynamic flow concepts versus dependence graphs, and in the way composite data types and pointers are represented. Korel and Laski treat components of composite data types as distinct variables, and invent names for dynamically allocated objects and pointers whereas Agrawal *et al.* 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.

*Concurrency.* Several methods for dynamic slicing of distributed programs have been proposed. Korel and Ferguson [89] and Duesterwald *et al.* [91] 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 [76] and Choi *et al.* [11, 12] do not consider this problem because the slices they compute are not executable programs. Duesterwald *et al.* [91] and Cheng [76] use *static* dependence graphs for computing *dynamic* slices. Although this is more space-efficient than the other approaches, 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 [11, 12].

## 7.3 *Applications*

Weiser [1] originally conceived of program slices as a model of the mental abstractions made by programmers when debugging a program, and advocated the use of slicing in debugging tools. The use of slicing for (automated) debugging was further explored by Lyle and Weiser [19], Choi *et al.* [12], Agrawal *et al.* [20], Fritzson *et al.* [21], and Pan and Spafford [22, 23]. Slicing has also proven to be of use for a variety of other applications including: parallelization [24], program differencing and integration [25, 26], software maintenance [27], testing [13, 28–30], reverse engineering [31–33], and compiler tuning [34]. Section 5 contains a detailed overview of how slicing is used in each of these application areas.

## 7.4 *Recent developments*

Two important characteristics of conventional slicing algorithms adversely affect the accuracy of program slices:

- The fact that slices consist of a subset of the original program's statements, sometimes with the additional constraint that a slice must be a syntactically valid program.
- The fact that slices are computed by tracing data and control dependences.

Section 6 discusses recent work by Field *et al.* [18] and by Ernst [50] for computing more accurate slices, where these 'restrictions' are removed. In essence, these slicing algorithms compute more accurate slices due to the use of compiler-optimization techniques, symbolic execution, and a variety of semantics-preserving transformations for eliminating spurious dependences. At the conceptual level, the algorithms by Field *et al.* and Ernst consist of the following components:

- Translation of a program to a suitable intermediate representation (IR).
- Transformation and optimization of the IR.
- Maintaining a mapping between the source text, the original IR, and the optimized IR.
- Extraction of slices from the IR.

Although Field *et al.* and Ernst have reported promising results, much work remains to be done in this area.

### 7.5 *Visualizing program slices*

Thus far, the emphasis of most slicing research has been on algorithmic aspects. Little attention has been paid to the question of how slices could best be visualized and interactively displayed or browsed [33, 50]. The recently developed SeeSlice tool of Ball and Eick [108] provides some interesting new ideas for visualizing slices of large programs.

## 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. The anonymous referees provided many constructive comments that I found extremely useful. Tom Reps provided the program and picture of Fig. 12. Susan Horwitz provided the program of Fig. 15. The programs shown in Figs 2 and 18 are adaptations of example programs that occur in Agrawal's dissertation [93]. The program of Fig. 19 was suggested by anonymous referee B.

This work was supported in part by the European Union under ESPRIT project #5399 (Compiler Generation for Parallel Machines—COMPARE).

## References

1. M. Weiser. *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method.* PhD thesis, University of Michigan, Ann Arbor, 1979.
2. M. Weiser. Programmers use slices when debugging. *Communications of the ACM*, **25** (1982) 446–452.
3. M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, **10** (1984) 352–357.
4. K. J. Ottenstein and L. M. Ottenstein. 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. *SIGPLAN Notices* **19**(5) (1984) 177–184.
5. D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure and M. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eighth ACM Symposium on Principles of Programming Languages*, 1981, pp. 207–218.
6. J. Ferrante, K. J. Ottenstein and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, **9** (1987) 319–349.

7. J.-F. Bergeretti and B. A. Carré. Information-flow and data-flow analysis of **while**-programs. *ACM Transactions on Programming Languages and Systems*, **7** (1985) 37–61.
8. T. Reps and T. Bricker. 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*, **17**(7) (1989).
9. B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, **29** (1988) 155–163.
10. R.M. Balzer. EXDAMS - Extendable Debugging And Monitoring System. In *Proceedings of the AFIPS SJCC*, **34** (1969) 567–5860.
11. B. P. Miller and J.-D. Choi. 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).
12. J.-D. Choi, B. P. Miller and R. H. B. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Transactions on Programming Languages and Systems*, **13** (1991) 491–530.
13. E. Duesterwald, R. Gupta and M. L. Soffa. Rigorous data flow testing through output influences. In *Proceedings of the Second Irvine Software Symposium ISS'92*, California, 1992, pp. 131–145.
14. M. Kamkar. *Interprocedural Dynamic Slicing with Applications to Debugging and Testing*. PhD thesis, Linkoping University, 1993.
15. G. A. Venkatesh. The semantic approach to program slicing. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pp. 107–119, 1991. *SIGPLAN Notices* **26**(6).
16. J. Q. Ning, A. Engberts and W. Kozaczynski. Automated support for legacy code understanding. *Communications of the ACM*, **37** (1994) 50–57.
17. J. Field and F. Tip. Dynamic dependence in term rewriting systems and its application to program slicing. In M. Hermenegildo and J. Penjam (eds) *Proceedings of the Sixth International Symposium on Programming Language Implementation and Logic Programming*, Vol. 844, pp. 415–431. Springer-Verlag, 1994.
18. J. Field, G. Ramalingam and F. Tip. Parametric program slicing. In *Conference Record of the Twenty-Second ACM Symposium on Principles of Programming Languages*, San Francisco, CA, 1995, pp. 379–392.
19. J. R. Lyle and M. Weiser. Automatic bug location by program slicing. In *Proceedings of the Second International Conference on Computers and Applications*, Beijing (Peking), China, 1987, pp. 877–883.
20. H. Agrawal, R. A. DeMillo and E. H. Spafford. Debugging with dynamic slicing and backtracking. *Software—Practice and Experience*, **23** (1993) 589–616.
21. P. Fritzson, N. Shahmehri, M. Kamkar and T. Gyimothy. Generalized algorithmic debugging and testing. *ACM Letters on Programming Languages and Systems*, **1** (1992) 303–322.
22. H. Pan. *Software Debugging with Dynamic Intrumentation and Test-Based Knowledge*. PhD thesis, Purdue University, 1993.
23. H. Pan and E. H. Spafford. Fault localization methods for software debugging. *Journal of Computer and Software Engineering*, 1994. To appear.
24. M. Weiser. Reconstructing sequential behaviour from parallel behaviour projections. *Information Processing Letters*, **17** (1983) 129–135.
25. S. Horwitz. 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*, pp. 234–245, White Plains, New York, 1990. *SIGPLAN Notices* **25**(6).
26. S. Horwitz, J. Prins and T. Reps. Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems*, **11** (1989) 345–387.

27. K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, **17** (1991) 751–761.
28. M. Kamkar, P. Fritzson and N. Shahmehri. Interprocedural dynamic slicing applied to interprocedural data flow testing. In *Proceedings of the Conference on Software Maintenance*, pp. 386–395, Montreal, Canada, 1993.
29. R. Gupta, M. J. Harrold and M. L. Soffa. An approach to regression testing using slicing. In *Proceedings of the Conference on Software Maintenance*, pp. 299–308, 1992.
30. S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages*, pp. 384–396, Charleston, SC, 1993.
31. J. Beck and D. Eichmann. Program and interface slicing for reverse engineering. In *Proceedings of the 15th International Conference on Software Engineering*, Baltimore, 1993, pp. 509–18.
32. D. Jackson and E. J. Rollins. A new model of program dependences for reverse engineering. In *Proceedings of the Second ACM SIGSOFT Conference on Foundations of Software Engineering*, New Orleans, LA, December 1994, pp. 2–10.
33. D. Jackson and E. J. Rollins. Abstraction mechanisms for pictorial slicing. In *Proceedings of the IEEE Workshop on Program Comprehension*, Washington, November 1994, pp. 82–88.
34. J. R. Larus and S. Chandra. Using tracing and dynamic slicing to tune compilers. Computer sciences technical report #1174, University of Wisconsin-Madison, 1993.
35. H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, pp. 246–256, 1990. *SIGPLAN Notices* **25**(6).
36. S. Horwitz, T. Reps and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, **12** (1990) 26–61.
37. B. Korel and J. Laski. Dynamic slicing of computer programs. *Journal of Systems and Software*, **13** (1990) 187–195.
38. A. Lakhotia. Graph theoretic foundations of program slicing and integration. Report CACS TR-91-5-5, University of Southwestern Louisiana, 1991.
39. R. Gupta and M. L. Soffa. A framework for generalized slicing. Technical report TR-92-07, University of Pittsburgh, 1992.
40. S. Horwitz and T. Reps. The use of program dependence graphs in software engineering. In *Proceedings of the 14th International Conference on Software Engineering*, pp. 392–411, Melbourne, Australia, 1992.
41. S. Horwitz, J. Prins and T. Reps. Integrating non-interfering versions of programs. In *Conference Record of the ACM SIGSOFT/SIGPLAN Symposium on Principles of Programming Languages*, pp. 133–145, 1988.
42. T. Reps and W. Yang. The semantics of program slicing and program integration. In *Proceedings of the Colloquium on Current Issues in Programming Languages*, Vol. 352 of *Lecture Notes in Computer Science*, pp. 60–74. Springer Verlag, 1989.
43. S. Horwitz and T. Reps. Efficient comparison of program slices. *Acta Informatica*, **28** (1991) 713–732.
44. M. Kamkar. An overview and comparative classification of static and dynamic program slicing. Technical Report LiTH-IDA-R-91-19, Linkoping University, Linkoping, 1991. To appear in *Journal of Systems and Software*.
45. T.J. Ball. *The Use of Control-Flow and Control Dependence in Software Tools*. PhD thesis, University of Wisconsin-Madison, 1993.
46. T. Ball and S. Horwitz. Slicing programs with arbitrary control-flow. In P. Fritzson, (ed.) *Proceedings of the First International Workshop on Automated and Algorithmic Debugging*, Vol. 749 of *Lecture Notes in Computer Science*, pp. 206–222. Springer-Verlag, 1993.

47. H. Agrawal. On slicing programs with jump statements. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pp. 302–312, Orlando, Florida, 1994. SIGPLAN Notices **29**(6).
48. J.-D. Choi and J. Ferrante. Static slicing in the presence of goto statements. *ACM Transactions on Programming Languages and Systems*, **16** (1994) 1087–1113.
49. R. Gopal. Dynamic program slicing based on dependence relations. In *Proceedings of the Conference on Software Maintenance*, pp.191–200, 1991.
50. M. Ernst. Practical fine-grained static slicing of optimized code. Technical Report MSR-TR-94-14, Microsoft Research, Redmond, WA, 1994.
51. W. Landi and B. G. Ryder. Pointer-induced aliasing: A problem classification. In *Proceedings of the Eighteenth ACM Symposium on Principles of Programming Languages*, pp. 93–103, January 1991.
52. W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the 1992 ACM Conference on Programming Language Design and Implementation*, pp. 235–248, San Francisco, 1992. *SIGPLAN Notices* **27**(7).
53. S. Horwitz, J. Prins and T. Reps. On the adequacy of program dependence graphs for representing programs. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pp. 146–157. ACM, 1988.
54. M. Weiser. Private communication, 1994.
55. H. K. N. Leung and H. K. Reghbati. Comments on program slicing. *IEEE Transactions on Software Engineering*, **SE-13** (1987) 1370–1371.
56. J. R. Lyle. *Evaluating Variations on Program Slicing for Debugging*. PhD thesis, University of Maryland, 1984.
57. P. Hausler. Denotational program slicing. In *Proceedings of the 22nd Hawaii International Conference on System Sciences*, pp. 486–494, Hawaii, 1989.
58. J. M. Barth. A practical interprocedural data flow analysis algorithm. *Communications of the ACM*, **21** (1978) 724–736.
59. J. Jiang, X. Zhou and D. J. Robson. Program slicing for C—the problems in implementation. In *Proceedings of the Conference on Software Maintenance*, pp. 182–190, 1991.
60. T. Reps. Private communication, 1994.
61. J. C. Hwang, M. W. Du and C. R. Chou. Finding program slices for recursive procedures. In *Proceedings of the 12th Annual International Computer Software and Applications Conference*, Chicago, 1988.
62. T. Reps. On the sequential nature of interprocedural program-analysis problems. Unpublished report, University of Copenhagen, 1994.
63. T. Reps, S. Horwitz, M. Sagiv and G. Rosay. Speeding up slicing. In *Proceedings of the Second ACM SIGSOFT Conference on Foundations of Software Engineering*, New Orleans, LA, December 1994, ACM SIGSOFT SE Notes, **19**(5), 11–20.
64. D. Binkley. Slicing in the presence of parameter aliasing. In *Proceedings of the Third Software Engineering Research Forum*, pp. 261–268, Orlando, Florida, November 1993.
65. A. V. Aho, R. Sethi and J. D. Ullman. *Compilers. Principles, Techniques and Tools*. Addison-Wesley, 1986.
66. J. P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Conference Record of the Sixth ACM Symposium on Principles of Programming Languages*, pp. 29–41, 1979.
67. T. Reps, M. Sagiv and S. Horwitz. Interprocedural dataflow analysis via graph reachability. Report DIKU TR 94-14, University of Copenhagen, Copenhagen, 1994.

68. D. Binkley. Precise executable interprocedural slices. *ACM Letters on Programming Languages and Systems*, **2** (1993) 31–45.
69. A. Lakhotia. Improved interprocedural slicing algorithm. Report CACS TR-92-5-8, University of Southwestern Louisiana, 1992.
70. K. B. Gallagher. *Using Program Slicing in Software Maintenance*. PhD thesis, University of Maryland, 1989.
71. J.-D. Choi, M. Burke and P. Carini. 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*, pp. 232–245. ACM, 1993.
72. S. Horwitz, P. Pfeiffer and T. Reps. 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).
73. H. Agrawal, R. A. DeMillo and E. H. Spafford. Dynamic slicing in the presence of unconstrained pointers. In *Proceedings of the ACM Fourth Symposium on Testing, Analysis, and Verification (TAV4)*, pp. 60–73, 1991. Also Purdue University technical report SERC-TR-93-P.
74. J. R. Lyle and D. Binkley. Program slicing in the presence of pointers. In *Proceedings of the Third Software Engineering Research Forum*, pp. 255–260, Orlando, Florida, November 1993.
75. W. E. Weihl. 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*, pp. 83–94, 1980.
76. J. Cheng. Slicing concurrent programs—a graph-theoretical approach. In P. Fritzson (ed) *Proceedings of the First International Workshop on Automated and Algorithmic Debugging*, Vol. 749 of *Lecture Notes in Computer Science*, pp. 223–240. Springer-Verlag, 1993.
77. A. Podgurski and L. A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, **16** (1990) 965–979.
78. D. Binkley. Interprocedural constant propagation using dependence graphs and a data-flow model. In P. A. Fritzson (ed.) *Proceedings of the 5th International Conference on Compiler Construction—CC '94*, Vol. 786 of *LNCS*, pp. 374–388, Edinburgh, UK, 1994.
79. F. Tip. *Generation of Program Analysis Tools*. PhD thesis, University of Amsterdam, 1995.
80. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, **13** (1991) 451–490.
81. R. Johnson, D. Pearson and K. Pingali. The program structure tree: Computing control regions in linear time. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pp. 171–185, Orlando, Florida, 1994. SIGPLAN Notices **29**(6).
82. B. Alpern, M. N. Wegman and F. K. Zadeck. Detecting equality of variables in programs. In *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages*, pp. 1–11, San Diego, 1988.
83. W. Yang, S. Horwitz and T. Reps. A program integration algorithm that accommodates semantics-preserving transformations. In *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments*, pp. 133–143, Irvine, CA, December 1990. *ACM SIGSOFT Software Engineering Notes* **15**(6).
84. K. D. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation*, pp. 57–66, Atlanta, Georgia, 1988. *SIGPLAN Notices* **23**(7).

85. D. E. Maydan, J. L. Hennessy and M. S. Lam. Efficient and exact data dependence analysis. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pp. 1–14, 1991. *SIGPLAN Notices* **26**(6).
86. H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. ACM Press Frontier Series. ACM Press, New York, 1991.
87. M. Kamkar, N. Shahmehri and P. Fritzson. Interprocedural dynamic slicing and its application to generalized algorithmic debugging. In *Proceedings of the International Conference on Programming Language Implementation and Logic Programming, PLILP '92*, 1992.
88. M. Kamkar, P. Fritzson and N. Shahmehri. Three approaches to interprocedural dynamic slicing. *Microprocessing and Microprogramming*, **38** (1993) 625–636.
89. B. Korel and R. Ferguson. Dynamic slicing of distributed programs. *Applied Mathematics and Computer Science*, **2** (1992) 199–215.
90. J. G. P. Barnes. *Programming in Ada*. 2nd edn. International Computer Science Series. Addison-Wesley, 1982.
91. E. Duesterwald, R. Gupta and M. L. Soffa. Distributed slicing and partial re-execution for distributed programs. In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, pp. 329–337, New Haven, Connecticut, 1992.
92. F. Tip. Generic techniques for source-level debugging and dynamic program slicing. In Proceedings of the Sixth International Joint Conference on Theory and Practice of Software Development, Aarhus, Denmark, May 1995, P. D. Mosses, M. Nielsen and M. I. Schwartzback (eds) (Springer Verlag) pp. 516–30.
93. H. Agrawal. *Towards Automatic Debugging of Computer Programs*. PhD thesis, Purdue University, 1991.
94. E. Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1982.
95. N. Shahmehri. *Generalized Algorithmic Debugging*. PhD thesis, Linkoping University, 1991.
96. T. Reps. Algebraic properties of program integration. *Science of Computer Programming*, **17** (1991) 139–215.
97. D. Binkley, S. Horwitz and T. Reps. Program integration for languages with procedure calls, 1994. Submitted for publication.
98. D. Binkley. Using semantic differencing to reduce the cost of regression testing. In *Proceedings of the IEEE Conference on Software Maintenance*, Orlando, Florida, November 1992.
99. L. M. Ott and J. J. Thuss. The relationship between slices and module cohesion. In *Proceedings of the 11th International Conference on Software Engineering*, pp. 198–204, 1989.
100. J. C. Hwang, M. W. Du and C. R. Chou. The influence of language semantics on program slices. In *Proceedings of the 1988 International Conference on Computer Languages*, Miami Beach, 1988.
101. M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, **13** (1991) 181–210.
102. J. Field. 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*, pp. 98–107, 1992. Published as Yale University Technical Report YALEU/DCS/RR-909.
103. J. W. Klop. Term rewriting systems. In S. Abramsky, D. Gabbay, and T. Maibaum (eds), *Handbook of Logic in Computer Science, Volume 2. Background: Computational Structures*, pp. 1–116. Oxford University Press, 1992.
104. H. P. Barendregt, M. C. J. D. van Eekelen, J. R. W. Glauert, J. R. Kennaway, M. J. Plasmeijer and M. R. Sleep. Term graph rewriting. In *Proc. PARLE Conference, Vol. II: Parallel Languages*, Vol. 259 of *Lecture Notes in Computer Science*, pp. 141–158, Eindhoven, The Netherlands, 1987. Springer-Verlag.

105. P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, **2** (1993) 176–201.
106. J. A. Bergstra, J. Heering and P. Klint (eds) *Algebraic Specification*. ACM Press Frontier Series. The ACM Press in cooperation with Addison-Wesley, 1989.
107. D. Weise, R. F. Crew, M. Ernst and B. Steensgaard. Value dependence graphs: Representation without taxation. In *Conference Record of the Twenty-First ACM Symposium on Principles of Programming Languages*, pp. 297–310, Portland, OR, 1994.
108. T. Ball and S. G. Eick. Visualizing program slices. In *Proceedings of the IEEE Symposium on Visual Languages*, pp. 288–295, St. Louis, Missouri, October 1994.