A Vocabulary of Program Slicing-Based Techniques

JOSEP SILVA, Universitat Politècnica de València

This article surveys previous work on program slicing-based techniques. For each technique, we describe its features, its main applications, and a common example of slicing using such a technique. After discussing each technique separately, all of them are compared in order to clarify and establish the relations between them. This comparison gives rise to a classification of techniques which can help to guide future research directions in this field.

Categories and Subject Descriptors: F.3.1 [**Logics and Meaning of Programs**]: Specifying and Verifying and Reasoning About Programs; D.3.1 [**Programming Languages**]: Formal Definitions and Theory

General Terms: Languages, Theory

Additional Key Words and Phrases: Program slicing, software engineering

ACM Reference Format:
Silva, J. 2012. A vocabulary of program slicing-based techniques. ACM Comput. Surv. 44, 3, Article 12 (June 2012), 41 pages.
DOI = 10.1145/2187671.2187674 http://doi.acm.org/10.1145/2187671.2187674

1. INTRODUCTION

Program slicing was originally introduced in 1984 by Mark Weiser [Weiser 1984]. Since then, many researchers have extended it in many directions and for all programming paradigms. The huge number of program slicing-based techniques has lead to the publication of different surveys [Tip 1995; Binkley and Gallagher 1996; Harman et al. 1996; Harman and Gallagher 1998; De Lucia 2001; Harman and Hierons 2001; Binkley and Harman 2004; Xu et al. 2005] trying to clarify the differences between them. However, each survey presents the techniques from a different perspective. For instance, some [Tip 1995; Binkley and Gallagher 1996; Harman and Gallagher 1998; De Lucia 2001; Harman and Hierons 2001; Xu et al. 2005] mainly focus on the advances and applications of program slicing-based techniques; in contrast, Binkley and Harman [2004] focus on their implementation by comparing empirical results, and Harman et al. [1996] try to compare and classify them in order to predict future techniques and applications. In this work, we follow the approach of Harman et al. [1996]. In particular, we compare and classify the techniques aiming at identifying relations between them in order to answer questions like "Is one technique a particular case of another? Is one technique more general or more expressive than another? Are they equivalent but expressed with different formalisms?"

This work has been partially supported by the Spanish Ministerio de Ciencia e Innovación under grant TIN2008-066220C03-02, by the Generalitat Valenciana under grant ACOMP/2010/042, and by the Universidad Politécnica de Valencia (Program PAID-06-08).
Author's address: Departamento de Sistemas Informáticos y Computación, Universitat Politècnica de València, Camino de Vera s/n, CP 46022, Valencia, Spain; email: jsilva@dsic.upv.es.

DOI 10.1145/2187671.2187674 http://doi.acm.org/10.1145/2187671.2187674

There is a well-developed theory and formalization of program slicing which allows for formally reasoning about the semantic implications of different slicing techniques (see e.g., [Venkatesh 1991; Giacobazzi and Mastroeni 2003; Binkley et al. 2004, 2006a, 2006b; Danicic et al. 2005; Ward and Zedan 2007]). This formal framework can be used, for example, to prove properties of a particular technique or of program slicing in general. For instance, Danicic et al. [2005] used the program schemas theory [Greibach 1985] in order to demonstrate that slices produced by standard algorithms are minimal for a class of programs. The formalization of the techniques presented and discussed here is not the objective of this article, but this theoretical background can be a valuable complement to this work. Therefore, interested readers that want to get deeper in a particular technique or are interested in its theoretical foundations are referred to those works.

This article has been structured in two parts: In the first part (Section 2), we describe all the techniques in a similar way to De Lucia [2001]. We propose a running example program and extract a slice from it by applying every technique; thus, the slices produced by each technique for the same program can be compared. In order to ensure it (because not all slicing criteria are comparable), we will produce at least one slice generated from a slicing criterion for each technique, which is comparable to the slicing criterion used in the other techniques. The main objective in this section is to discuss each technique separately and compare it with the others by temporarily delimiting it and showing its peculiarities and main applications. Here, we do not focus on *how* the slices are produced but on *what* slices are produced. Therefore, we will not explain the algorithms or the internal mechanisms for producing the slices of each technique. Rather, we will discuss for each technique what information is needed to produce a slice, what peculiarities this slice has with respect to other techniques, and what its main applications are. Of course, we will refer the interested reader to the sources where the techniques were defined and where deeper details about their implementation can be found.

This part may be useful for a researcher (not necessarily a program slicing expert) who is looking for a program slicing technique and she wants to know which of the wide variety of slicing techniques better fits her needs. To ease the search, we summarize the goal of each technique with a single question and list its main applications.

In the second part (Section 3), we revisit the classification introduced by Harman et al. [1996] and extend it with new slicing techniques and dimensions. This analysis provides useful information that allows us to classify the slicing techniques and establish relations between them. In particular, we relate all the techniques of the first part by identifying three kinds of relations between them: generalization, superset, and composition. With these relations, we produce a graph of slicing techniques where the relations between them establish a hierarchy. With the information provided by the study, we try to predict new slicing techniques not published yet. Finally, Section 4 concludes.

2. PROGRAM SLICING TECHNIQUES

2.1. Program Slicing [Weiser 1984]

The original ideas of program slicing come from the Ph.D. dissertation of Mark Weiser [Weiser 1979] that were presented in the *Proceedings of the International Conference on Software Engineering* [1981] and finally published in *Transactions on Software Engineering* [1984].

Program slicing is a technique for decomposing programs by analyzing their data and control flow. Roughly speaking, a *program slice* consists of those program statements

```
*(1)  read(text);
 (2)  read(n);
 (3)  lines = 1;
 (4)  chars = 1;
 (5)  subtext = "";
 (6)  c = getChar(text);
 (7)  while (c != '\eof')
 (8)       if (c == '\n')
 (9)       then lines = lines + 1;
*(10)           chars = chars + 1;
 (11)      else chars = chars + 1;
 (12)           if (n != 0)
*(13)           then subtext = subtext ++ c;
 (14)                n = n - 1;
 (15)      c = getChar(text);
 (16) write(lines);
 (17) write(chars);
*(18) write(subtext);
```

(a) Example program

```
(1)  read(text);

(3)  lines = 1;



(6)  c = getChar(text);
(7)  while (c != '\eof')
(8)       if (c == '\n')
(9)       then lines = lines + 1;









(15)      c = getChar(text);
(16) write(lines);
```

(b) Slice with respect to $\langle 16, lines \rangle$

Fig. 1. Example of a slice.

which are (potentially) related to the values computed at some program point and/or variable, referred to as a *slicing criterion*.

As it was originally defined,

> A slice is itself an executable program subset of the program whose behavior must be identical to the specified subset of the original program's behavior.

However, the constraint of being an executable program has sometimes been relaxed. Given a program p, slices are produced with respect to a given slicing criterion $\langle s, v \rangle$ which specifies a statement s and a set of variables v in p. Note that the variables in v do not necessarily appear in s; consider, for instance, the slicing criterion $\langle 18, \{lines, chars, subtext\} \rangle$ for the program in Figure 1(a). Observe that in this case, not all the variables appear in line 18.

As an example of a slice, consider the program in Figure 1(a) (for the time being, the user can ignore the breakpoints marked with $*$), where function `getChar` extracts the first character of a string. This program is an augmented version of the UNIX *word-count* program: it takes a text (i.e., a string of characters including '\n'—carriage return—and '\eof'—end-of-file—) and a number n, and it returns the number of characters and lines of the text and a subtext composed of the first n characters excluding '\n'. We need to augment the functionality of word-count because, in the following, this will be our running example that we will use with all the techniques in order to compare their slices produced; and for some techniques, word-count is not enough to show their differences with respect to other techniques. A slice of this program with respect to the slicing criterion $\langle 16, lines \rangle$ is shown in Figure 1(b).

Together with the running example, we will use a *running slicing criterion* with all the techniques in order to be able to compare their slices produced with respect to similar slicing criteria. Nevertheless, the shape of the slicing criterion changes from one technique to the other. Therefore, we cannot use the same slicing criterion in their comparison. What we will do is use comparable slicing criteria. Intuitively, two slicing criteria s_1 and s_2 are comparable if for all parameter p (such that $p \in s_1$ and $p \in s_2$), the value of p is the same in both slicing criteria. In Section 3, we will precisely specify all the parameters and their values that can be used in current slicing techniques.

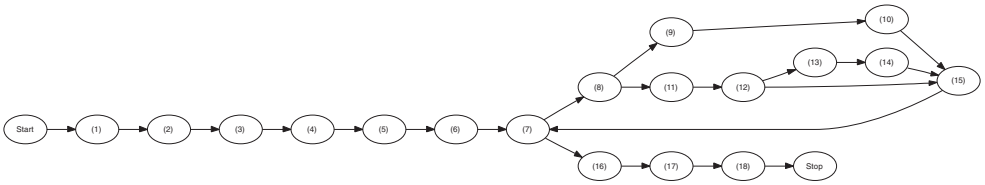


Fig. 2. Control flow graph of Figure 1 (a).

2.2. Static Slicing [Weiser 1984]

The original definition of program slicing was static [Weiser 1984] in the sense that it did not consider any particular input for the program being sliced. Particularly, the slice shown in Figure 1(b) is a static slice with respect to the slicing criterion $\langle 16, lines \rangle$. It is called static because it does not consider any particular execution (i.e., it works for any possible input data).

In order to extract a slice from a program, the dependencies between its statements must be computed first. The *control flow graph* (CFG) is a data structure which makes the control dependencies for each operation in a program explicit. For instance, the CFG of the program in Figure 1(a) is depicted in Figure 2.

However, the CFG does not suffice for computing program slices, because it only stores control dependencies and data dependencies are also necessary. For this reason, the CFG must be annotated with data flow information by marking the set of variables defined and referenced at each node. A detailed explanation of how to annotate CFGs and how to extract a slice from annotated CFGs can be found in Binkley and Gallagher [1996]. Ottenstein and Ottenstein [1984] noted that the *program dependence graph* (PDG) [Kuck et al. 1981; Ferrante et al. 1987] was the ideal data structure for program slicing because it allows us to build slices in linear time on the number of nodes of the PDG.[1] PDGs make explicit both the data and control dependencies for each operation in a program. In essence, a PDG is an oriented graph where the nodes represent statements in the source code the and edges represent control and data flow dependencies between statements in such a way that they induce a partial ordering in the nodes, preserving the semantics of the program. As an example, the PDG of the program in Figure 1(a) is depicted in Figure 3 where solid arrows represent control dependencies and dotted arrows represent flow dependences. Here, nodes are labeled with the number of the statement they represent, except node *Start*, which represents the start of the program. The solid arrow between nodes 7 and 8 indicates that the execution of statement 8 depends on the execution of statement (7). The same happens with the solid arrow between statements (8) and (12); thus, transitively, the execution of statement (12) also depends on the execution of statement (7).

Question answered. What program statements can influence these variables at this statement?

Main applications. Program comprehension, debugging, algorithmic debugging, dead code removal, program segmentation, program analysis, software quality assurance, program differencing, software maintenance, testing, program parallelization, module cohesion analysis, partial evaluation, and program integration.

[1]Although the cost of computing a slice from a PDG of N nodes is $O(N)$, the cost of building the PDG is $O(N^2)$. See Ferrante et al. [1987] for details.

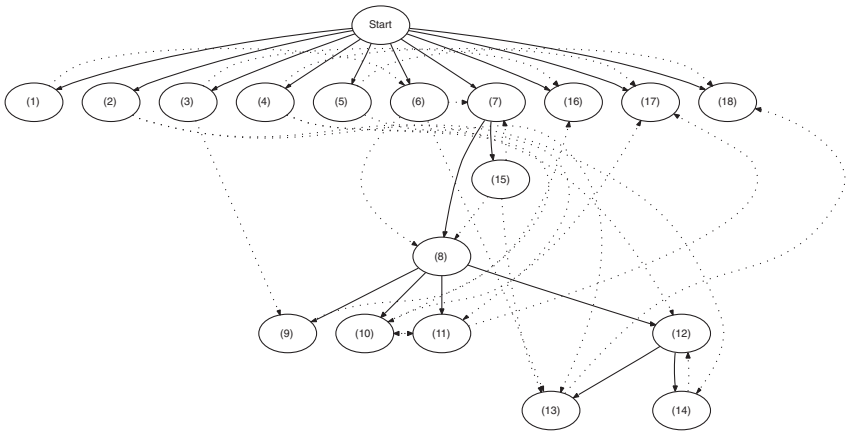


Fig. 3. Program dependence graph of Figure 1(a).

2.3. Dynamic Slicing [Korel and Laski 1988]

One of the main applications of program slicing is debugging. Often, during debugging, the value of a variable v at some program statement s is observed to be incorrect. A program slice with respect to $\langle s, v\rangle$ contains the cause of the error.

However, in such a case, we are interested in producing a slice formed by those statements that could cause this particular error, that is, we are only interested in one specific execution of the program. Such slices are called dynamic slices [Korel and Laski 1988], since they use dynamic information during the process. In general, dynamic slices are much smaller than static ones because they contain the statements of the program that affect the slicing criterion for a particular execution (in contrast to any execution, as happens with static slicing).

During a program execution, the same statement can be executed several times in different contexts (i.e., with different values of the variables); as a consequence, pointing out a statement in the program is not enough in dynamic slicing. A dynamic slicing criterion needs to specify which particular execution of the statement during the computation is of interest; thus it is defined as $\langle s^i, v, \{a_1, \ldots, a_n\}\rangle$, where i is the position of statement s in the execution history[2]; v is the set of variables we are interested in, and set $\{a_1, \ldots, a_n\}$ compares the initial values of the program's input. As an example, consider the input values $\{text = \text{“}hello\ world!\backslash eof\text{”}, n = 4\}$ for the program in Figure 1(a). The execution history would be

$$(1, 2, 3, 4, 5, 6, 7, (8, 11, 12, 13, 14, 15, 7)^{12}, 16, 17, 18),$$

where the superscript 12 indicates that the statements inside the parenthesis are repeated twelve times in the execution history.

A dynamic slice of the program in Figure 1(a) with respect to the slicing criterion $\langle 16^{92},\ \{lines\},\ \{text = \text{“}hello\ world!\backslash eof\text{”}, n = 4\}\rangle$ is shown in Figure 4. Note that this slice is much smaller than its static counterpart, because here, the slicer can compute the specific control and data flow dependencies produced by the provided input data. However, it comes with a cost: the computation of such dependencies usually implies the

[2]This notation has been used in two different ways in the literature. We use the original definition as defined in Korel and Laski [1988] and later used, for instance, in Tip [1995]. However, there is another meaning for s^i, which stands for the statement s when it is executed the ith time; as defined in Agrawal and Horgan [1990] and later user, for instance, in Binkley and Gallagher [1996]".

```
(3)  lines = 1;
(16) write(lines);
```

Fig. 4. Dynamic slice of Figure 1(a) with respect to $\langle 16^{92},\ \{lines\},\ \{text = \text{“}hello\ world!\backslash eof\text{”}, n = 4\}\rangle$.

computation of an expensive (measured in time and space) and complex data structure (e.g., a trace [Sparud and Runciman 1997]).

In their definition of dynamic slicing, Korel and Laski [1998] introduced one important novelty: their slices must follow identical paths as their associated programs. Roughly, this means that the original program and the slice must produce the same execution history before they reach the slicing criterion, except for the statements not influencing the slicing criterion.

Definition 2.1 (*Korel and Laski's Dynamic Slice [1988]*). Let $c = (s^i, V, x)$ be a slicing criterion of a program p and T the trajectory of p on input x. A dynamic slice of p on c is any executable program p' that is obtained from p by deleting zero or more statements, such that when executed on input x, it produces a trajectory T' for which there exists an execution position i' such that the following hold.

(1) $Front(T', i') = DEL(Front(T, i), T(j) \notin N' \wedge 1 \leq j \leq q)$.
(2) For all $v \in V$, the value of v before the execution of instruction $T(i)$ in T equals the value of v before the execution of instruction $T'(i')$ in T'.
(3) $T'(i') = T(i) = s$,

where N' is a set of instructions in p'; $Front(T, j)$ returns the first j elements of sequence T from 1 to j inclusive; and $DEL(T, \pi)$ is a filtering function which takes a predicate π and returns the trajectory obtained by deleting from T the elements that satisfy π.

This property constitutes a new way to compute slices that can be combined with many other forms of slicing (e.g., Binkley et al. [2006a]) and that determines whether or not dynamic slicing is subsumed by other forms of slicing. Therefore, the path-aware condition should be fixed in order to compare two slicing techniques (whatever they are). In the rest of the article—unless the contrary is stated—we will always consider that slicing techniques are path-unaware. Therefore, subsequent references to dynamic slicing will really refer to path-unaware dynamic slicing (thus, a different algorithm than the one defined by Korel and Laski [1988], see Agrawal and Horgan [1990]). We refer the reader to Section 3 for a description of slicing dimensions and their use for comparing slicing techniques.

Question answered. For this particular execution, what program statements can influence these variables at this statement?

Main applications. Debugging, testing, and tuning compilers.

2.4. Backward Slicing [Weiser 1984]

A program can be traversed forwards or backwards from the slicing criterion. When we traverse it backwards (backward slicing), we are interested in all those statements that could influence the slicing criterion. In contrast, when we traverse it forwards (forward slicing), we are interested in all those statements that could be influenced by the slicing criterion.

The previously described original method by Weiser was static backward slicing. The main applications of backward slicing are debugging, program differencing, and testing. As an example, the slice shown in Figure 1(b) is a backward slice of the program in Figure 1(a).

```
(3)  lines = 1;
(9)       lines = lines + 1;
(16) write(lines);
```

Fig. 5. Forward static slice of Figure 1(a) with respect to $\langle 3, \{lines\}\rangle$.

```
(3)  lines = 1;
(9)       lines = lines + 1;
```

Fig. 6. Chop of Figure 1(a) with respect to $\langle source = \{(3, \{lines\})\}, sink = \{(9, \{lines\})\}\rangle$.

Question answered. What program statements can influence these variables at this statement?

Main applications. Program comprehension, debugging, algorithmic debugging, dead code removal, program segmentation, program analysis, program differencing, software maintenance, testing, program parallelization, module cohesion analysis and program integration.

2.5. Forward Slicing [Bergeretti and Carré 1985]

Forward slicing [Bergeretti and Carré 1985] allows us to determine how a modification in a part of the program will affect other parts of the program. As a consequence, it has been used for dead code removal and for software maintenance. In this context, Reps and Bricker were the first to use the notion of forward slicing [1989]. However, despite backward slicing being the preferred method for debugging, forward slicing has also been used for this purpose. In particular, forward slicing can detect initialization errors [Gaucher 2003].

A forward static slice of our running example with respect to the previously used slicing criterion $\langle 16, \{lines\}\rangle$ would only contain statement (16) because *lines* is not used after line (16).

A more clarifying example is the forward static slice of the program in Figure 1(a) with respect to the slicing criterion $\langle 3, \{lines\}\rangle$, as shown in Figure 5.

Question answered. What program statements can be influenced by these variables at this statement?

Main applications. Program differencing, debugging, program comprehension, program analysis, dead code removal, software maintenance, and testing.

2.6. Chopping [Jackson and Rollins 1994]

In chopping [Jackson and Rollins 1994], the slicing criterion selects two sets of variables, *source* and *sink*, and then it computes all the statements in the program that being affected by source, affect sink. Therefore, chopping is a generalization of both forward and backward slicing where either source or sink is empty. As noted by Reps and Rosay [1995], chopping is particularly useful for detecting those statements that transmit effects from one part of the program (source) to another (sink).

For instance, a chop of the program in Figure 1(a) with respect to the slicing criterion $\langle source = \{(3, \{lines\})\}, sink = \{(16, \{lines\})\}\rangle$ is shown in Figure 5. Similarly, a chop of the program in Figure 1(a) with respect to the slicing criterion $\langle source = \{(3, \{lines\})\}, sink = \{(9, \{lines\})\}\rangle$ is shown in Figure 6.

Question answered. What program statements can influence these variables at this statement while they are influenced by these (other) variables at this other statement?

Main applications. Program analysis and debugging.

```
(3)  lines = 1;
(6)  c = getChar(text);
(8)       if (c == '\n')
(16) write(lines);
```

Fig. 7. Relevant slice of Figure 1(a) with respect to $\langle 16^{92}, \{lines\}, \{text = \text{“}hello\ world!\backslash eof\text{”}, n = 4\}\rangle$.

2.7. Relevant Slicing [Agrawal et al. 1993]

A dynamic program slice only contains the program statements that actually affect the slicing criterion. However, it is sometimes interesting to include in the slice those statements that could have affected the slicing criterion (e.g., in debugging). This is the objective of relevant slicing [Agrawal et al. 1993], which computes all the statements that could potentially affect the slicing criterion.

A slicing criterion for relevant slicing is exactly the same as for dynamic slicing, but if we compute a relevant slice and a dynamic slice with respect to the same slicing criterion, the relevant slice is a superset of the dynamic slice, because it contains all the statements of the dynamic slice and also contains those statements of the program that did not affect the slicing criterion but could have affected it if they would have changed (for instance, because they were faulty).

Let us explain it with an example: consider the previously used slicing criterion $\langle 16^{92}, \{lines\}, \{text = \text{“}hello\ world!\backslash eof\text{”}, n = 4\}\rangle$. The relevant slice computed is shown in Figure 7. It contains all the statements of the dynamic slice (see Figure 4) and also includes statements `(6)` and `(8)`. Statement `(6)` could have influenced the value of `lines` being redefined to “`(6) c = '\n';`” and statement `(8)` could have influenced the value of `lines` being redefined to “`(8) if (c != '\n')`”.

It should be clear that the slices produced in relevant slicing can be non-executable. The reason being that the main application of such a technique is debugging, where the programmer is interested in those parts of the program that can contain the bug. The statements included in the slice are those whose contamination could result in the contamination of the variable of interest. In order to make the slice executable, preserving the behavior (including termination) of the original program it is necessary to augment the slice with those statements that are required for the evaluation of all the expressions included in the slice (even if this evaluation does not influence the variable of interest).

The forward version of relevant slicing [Gyimóthy et al. 1999] can be useful in debugging and in program maintenance. A forward relevant slice contains those statements that could be affected by the slicing criterion (if it is redefined). Therefore, it could be used to study module cohesion by determining what could be the impact of a module modification over the rest of modules.

Question answered. What program statements could influence these variables at this statement if they were redefined?

Main applications. Debugging and program maintenance.

2.8. Hybrid Slicing [Gupta and Soffa 1995]

When debugging, it is usually interesting to work with dynamic slices because they focus on a particular execution (i.e., the one that showed a bug) of the program being debugged; therefore, they are much more precise than static ones. However, computing dynamic slices is very expensive in time and space due to the large data structures (up to gigabytes) that need to be computed. In order to increase the precision of static slicing without incurring the computation of these large structures needed for dynamic slicing, a new technique called hybrid slicing [Gupta and Soffa 1995] was proposed. A

```
(1)  read(text);
(4)  chars = 1;
(6)  c = getChar(text);
(7)  while (c != '\eof')
(8)       if (c == '\n')
(11)      then chars = chars + 1;
(15)      c = getChar(text);
(17) write(chars);
```

Fig. 8. Hybrid slice of Figure 1(a) with respect to $\langle 17, \{chars\}, \{1, 13, 13, 13, 18\}\rangle$.

hybrid slice is more accurate—and consequently smaller—than a static one and less costly than a dynamic slice.

The key idea of hybrid slicing consists of integrating dynamic information into the static analysis. In particular, the information provided to the slicer is a set of breakpoints inserted into the source code that can be activated when the program is executed, thus providing information about which parts of the code have been executed. This information allows the slicer to eliminate from the slice all the statements which are in a non-executed possible path of the computation.

For instance, consider the program in Figure 1(a) which contains four breakpoints marked with '$*$'. A particular execution will activate some of the breakpoints; this information can be used by the slicer. For instance, a possible execution could be $\{1, 13, 13, 13, 10, 13, 18\}$, which means that the loop has been entered five times, and both branches of the outer if-then-else have been executed.

A hybrid slicing criterion is a triple $\langle s, v, \{b_1, \ldots, b_n\}\rangle$, where s and v have the same meaning as in static slicing, and the set $\{b_1, \ldots, b_n\}$ is the sequence of breakpoints activated during a particular execution of the program.

Figure 4 shows the hybrid slice of our running example with respect to the slicing criterion $\langle 16, \{lines\}, \{1, 13, 13, 13, 18\}\rangle$. Figure 8 shows the hybrid slice with respect to the slicing criterion $\langle 17, \{chars\}, \{1, 13, 13, 13, 18\}\rangle$.

Question answered. For the set of executions defined by this set of breakpoints, what program statements can influence these variables at this statement?

Main applications. Debugging.

2.9. Intraprocedural Slicing *(Weiser, 1984)*

The original definition of program slicing has been later classified as *intraprocedural slicing* (i.e., the slice in Figure 1(b) is an intraprocedural slice), because the original algorithm did not take into account information related to the fact that slices can cross the boundaries of procedure calls. In such cases, it generates wrong criteria which are not feasible in the control flow of the program.

This does not mean that the original definition fails to slice multiprocedural programs; it means that it loses precision in such cases.

As an example, consider the program in Figure 9. A static backward slice of this program with respect to the slicing criterion $\langle 16, \{x\}\rangle$ includes all the statements of the program except statements (11), (12), and (13). However, it is clear that statements (3) and (8) included in the slice cannot affect the slicing criterion. They are included in the slice because procedure `sum` influences the slicing criterion, and statements (3) and (8) can influence procedure `sum`. However, they cannot transitively affect the slicing criterion. In particular, the problem of Weiser's algorithm is that the call `sum(x,1)` causes the slice to go down into procedure `sum`, and then it goes up to all the calls to `sum`, including the irrelevant call `sum(lines,1)`. This problem is due to the fact that Weiser's algorithm does not keep information about the calling context when it

```
(1)  program main
(2)  read(text);
(3)  lines = 1;
(4)  chars = 1;
(5)  c = getChar(text);
(6)  while (c != '\eof')
(7)       if (c == '\n')
(8)       then sum(lines,1);
(9)       else increment(chars);
(10)      c = getChar(text);
(11) write(lines);
(12) write(chars);
(13) end

(14) procedure increment(x)
(15) sum(x,1);
(16) return

(17) procedure sum(a, b)
(18) a = a + b;
(19) return
```

Fig. 9. Example of a multi-procedural program.

traverses procedures up and down. A detailed explanation of this problem can be found in Gallagher [2004].

This loss of precision has been later solved by another technique called *interprocedural slicing* (see Section 2.10).

Question answered. What program statements can influence these variables at this statement?

Main applications. Program comprehension, debugging, algorithmic debugging, dead code removal, program segmentation, program analysis, software quality assurance, program differencing, software maintenance, testing, program parallelization, module cohesion analysis, partial evaluation, and program integration.

2.10. Interprocedural Slicing [Horwitz et al. 1988]

In 1988, Horwitz et al. noted that the program dependence graph was not appropriate for representing multiprocedural programs, and they proposed a new dependence graph representation of programs called *system dependence graph* (SDG) [1988]. This new representation incorporates collections of procedures with procedure calls, and it allows us to produce more precise slices from multiprocedural programs, because it has information available about the actual procedures' calling context.

For instance, consider `Program 1` in Figure 24 together with the slicing criterion $\langle 3, \{chars\}\rangle$. An intraprocedural static slice contains exactly the same statements (`Program 1`). However, it is clear that procedure `increment` cannot affect the slicing criterion (in fact, it is dead code). Roughly speaking, this inaccuracy is due to the fact that the call to procedure `sum` makes the slice include this procedure. Then, all the calls to this procedure can influence it, and thus, procedure `increment` is also included.

In contrast, an interprocedural static slice (`Program 2`) uses information stored in the system dependence graph about the calling context of procedures. Hence, it would remove procedure `increment`, thus increasing the precision with respect to the intraprocedural algorithms.

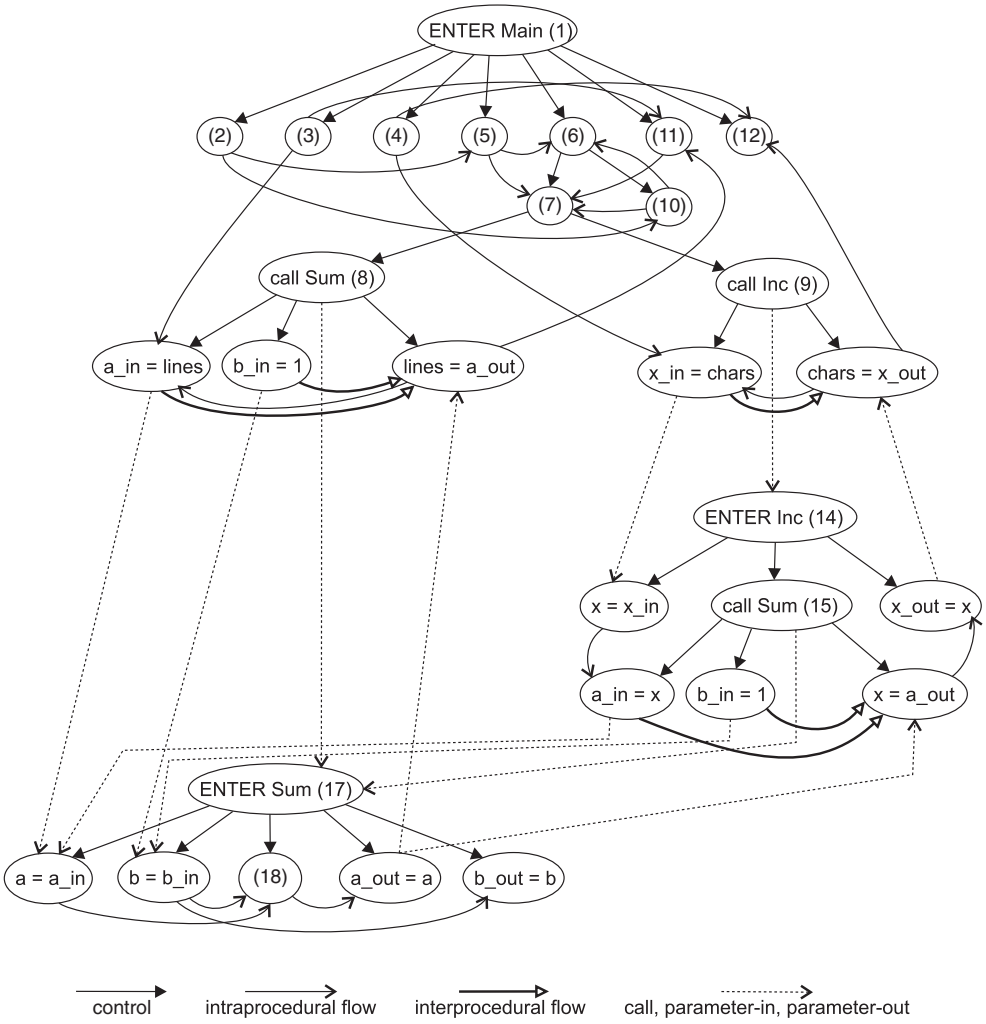


Fig. 10. System dependence graph of the program in Figure 9.

In order to understand what additional information the SDG provides about the calling context of procedures, we can observe the SDG of Figure 9 shown in Figure 10.

The SDG of Figure 10 contains all the information needed to produce precise interprocedural slices. As the PDG, it contains the control and flow relations of the program, but it also contains other relations: interprocedural flow relations are represented by bold arrows, and call, parameter in, and parameter out relations are represented by dotted arrows. Moreover, new nodes are included to represent the values of the parameters of all procedures when entering the procedure (*_in* parameters: information going down) and when leaving the procedure (*_out* parameters: information going up). Together with the definition of the SDG, Horwitz et al. [1988] introduced an algorithm to produce precise interprocedural slices.

Later, Gallagher [2004] proposed an alternative solution to solve the imprecision of Weiser's algorithm with interprocedural programs. He showed that it is possible to produce precise interprocedural slices by using the PDG together with a call graph.

```
(1)  program main
(2)  read(text);
(4)  chars = 1;
(5)  c = getChar(text);
(6)  while (c != '\eof')
(7)       if (c == '\n')
(9)       else increment(chars);
(10)      c = getChar(text);

(14) procedure increment(x)
(15) sum(x,1);
(16) return

(17) procedure sum(a, b)
(18) a = a + b;
(19) return
```

Fig. 11. Interprocedural slice of Figure 9 with respect to $\langle 16, \{x\}\rangle$.

```
(...)
(1)  Person *p;
(2)  if (x>0)
(3)  then p = new Person();
(4)  else p = new Worker();
(...)
```

Fig. 12. Object-oriented code.

Figure 11 shows the interprocedural slice of the program in Figure 9, with respect to the slicing criterion $\langle 16, \{x\}\rangle$.

2.10.1. Object-Oriented Slicing [Larsen and Harrold 1996]. Object-oriented programs introduce additional features, such as classes, objects, inheritance, polymorphism, instantiation, etc, which cannot be handled with standard SDGs. Therefore, in order to slice object-oriented programs, the SDG must be extended. Larsen and Harrold [1996] proposed an extension of the SDG for object-oriented programs which can still use the standard SDG algorithm. In the SDGs proposed by Larsen and Harrold, each single class is represented by a *class dependence graph* (ClDG) which represents control and data dependencies in a class without knowledge of calling environments. ClDGs can be reused in the presence of inheritance. That is, derived classes are built by constructing a representation of new methods and reusing the representation of inherited methods.

ClDGs take into account instantiation and polymorphism. When class $C1$ instantiates class $C2$ (e.g., through a declaration or by using the operator **new**), there is an implicit call to $C2$'s constructor, which must be represented in the ClDG. Similarly, polymorphic method calls introduce an additional complexity which must be solved in the ClDGs. For instance, consider the piece of code in Figure 12 where class `Worker` extends class `Person`.

Under the assumption that `x` is a parameter of the program, it is not possible to know at compilation time whether `p` will instantiate `Worker` or `Person`. Static analysis must consider both possibilities. To solve this situation, the ClDG uses a special vertex called *polymorphic choice vertex* to represent the dynamic choice among the possible destinations of the method calls.

Another particularity of object-oriented programs is that variable references are often replaced with method calls that simply return the value of the variable. Therefore, in order to allow the user to slice on the values returned by a method, the slicing

```
(1)  read(text);
(5)  subtext = "";
(18) write(subtext);
```

Fig. 13. Quasi-static slice of Figure 1(a) with respect to $\langle 18, \{subtext\}, \{n = 0\}\rangle$.

criterion is slightly generalized. A static slicing criterion for an object-oriented program is a pair $\langle s, m\rangle$, which specifies a statement s and a variable or a method call m. If m is a variable, it must be defined or used at s; if m is a method call, it must be called at s.

2.10.2. Aspect-Oriented Slicing [Zhao 2002]. Aspect-oriented programs incorporate new concepts and associated constructs, namely join points, pointcut, advice, introduction, and aspect. To cope with them, Zhao [2002] proposed a new extension of the SDG called *aspect-oriented system dependence graph* (ASDG).

The ASDG is constructed by constructing the SDG of the non-aspect code of the program, constructing dependence graphs for the aspect code of the program, and connecting the graphs by adding special vertices and arcs. The result is the ASDG which can be used in combination with the standard SDG algorithm.

Question answered. What program statements can influence these variables at this statement?

Main applications. Program comprehension, debugging, algorithmic debugging, dead code removal, program segmentation, program analysis, software quality assurance, program differencing, software maintenance, testing, program parallelization, module cohesion analysis, partial evaluation, and program integration.

2.11. Quasi-Static Slicing [Venkatesh 1991]

While static slicing computes slices with respect to any execution, dynamic slicing computes slices with respect to a particular execution. However, it is sometimes interesting to produce a slice with respect to a particular set of executions (e.g., in program understanding). Quasi-static slicing [Venkatesh 1991] can be used in those applications in which a set of the program inputs are fixed, and the rest of the inputs is unknown. This leads to a potentially infinite set of considered executions.

A quasi-static slicing criterion is a tuple $\langle s, v, \{a_1, \ldots, a_m\}\rangle$ where s and v have the same meaning as in static slicing, and the set $\{a_1, \ldots, a_m\}$ is a mapping from (some of the) input variables to values. For instance, a quasi-static slicing criterion for the program in Figure 1(a) could be $\langle 18, \{subtext\}, \{n = 0\}\rangle$. The slice computed with respect to this criterion is shown in Figure 13.

In our running example, the slice produced with respect to the slicing criterion $\langle 16, \{lines\}, \{text = \text{“}hello\ world!\backslash eof\text{”}\}\rangle$ would produce the slice shown in Figure 4.

Question answered. For the set of executions in which these inputs have these values, what program statements can influence these variables at this statement?

Main applications. Debugging and program comprehension.

2.12. Call-Mark Slicing [Nishimatsu et al. 1999]

Given a dynamic slicing criterion $\mathcal{D}$ which is comparable to a static slicing criterion $\mathcal{S}$, the minimal slice produced for $\mathcal{D}$ is smaller or equal to the minimal slice produced for $\mathcal{S}$. However, when slicing real programs, the minimal slice produced for $\mathcal{D}$ is generally much smaller than the minimal slice produced for $\mathcal{S}$ [Binkley et al. 2006a; Takada et al. 2002].

Nevertheless, this reduction of the size comes with a cost: computing dynamic slices with standard algorithms is much more expensive than computing their static

```
(3)  lines = 1;
(11) write(lines);
```

Fig. 14. Call-mark slice of Figure 9 with respect to $\langle 13, \{lines\}\rangle$.

counterparts. The reason being that computing an execution trace is necessary to extract the dependences between actually executed statements.

Nishimatsu et al. [1999] proposed a new slicing technique named call-mark slicing which allows us to reduce the cost of constructing dynamic slices though reducing their precision. The objective is to establish a compromise between static slicing and dynamic slicing in the following sense: call-mark slices are generally smaller than static slices, but they are less expensive to build than dynamic slices. To do so, this technique uses dynamic information when constructing the PDG. A call-mark slicing criterion is exactly the same as a static slicing criterion but augmented with a complete input. Hence, $\langle s, v, \{a_1, \ldots, a_n\}\rangle$, where s is a statement, v is the set of variables we are interested in, and the set $\{a_1, \ldots, a_n\}$ is the initial values of the program's input. The main difference between this technique and dynamic slicing is the way of using the dynamic information. Call-mark slicing uses the dynamic information to determine whether or not each execution/procedure call statement in the program is executed. These call statements are marked. Then, this information is used to prune the PDG by removing those statements not marked as executed. The result is a more precise PDG which can be traversed with standard static techniques.

As an example, if we consider the program in Figure 9 and the slicing criterion $\langle 13,\ \{lines\}, \{text = \text{“}hello\ world!\backslash eof\text{”}, n = 4\}\rangle$, we get the call nodes of the PDG (9) and (15) as marked. Because node (8) is not marked, this statement is removed from the slice (despite it statically affecting variable *lines* at statement (13)). The call-mark slice produced is shown in Figure 14. Note that in this case, we where so lucky that the call-mark slice is as accurate as the corresponding dynamic slice; but, with real programs, this is not usually the case.

Question answered. For this particular execution, what program statements can influence these variables at this statement?

Main applications. Debugging, program comprehension, and testing.

2.13. Dependence-Cache Slicing [Takada et al. 2002]

After the first attempt with call-mark slicing to define a slicing technique which uses dynamic information to prune the PDG, Takada et al. [2002] proposed a new slicing technique named dependence-cache slicing which also allows us to prune the PDG with dynamic information. Similarly to call-mark slicing, dependence-cache slicing uses the dynamic information to build a more precise PDG. In this new PDG (called PDG_{DS} [Takada et al. 2002]), the data-dependence relations are those which are possible with the input data provided. Therefore, many infeasible paths are removed from the PDG. While their objective is the same, call-mark slicing and dependence-cache slicing are very different in the way they prune the PDG. On the one hand, call-mark slicing deletes nodes from the PDG. On the other hand, dependence-cache slicing deletes edges. In particular, dependence-cache slicing constructs the PDG_{DS} in two steps: (i) a standard PDG is constructed without adding data dependence edges; and then, (ii) a data-dependencies collection algorithm is used to add data edges to the PDG_{DS}, which are feasible in the considered execution.

For instance, the PDG_{DS} of our running example constructed from the input data $\{text = \text{“}hello\ world!\backslash eof\text{”}, n = 4\}$ would be exactly the same as the PDG in Figure 3 but removing the data dependence edges (3)→(9), (4)→(10), (9)→(16), (10)→(11), and

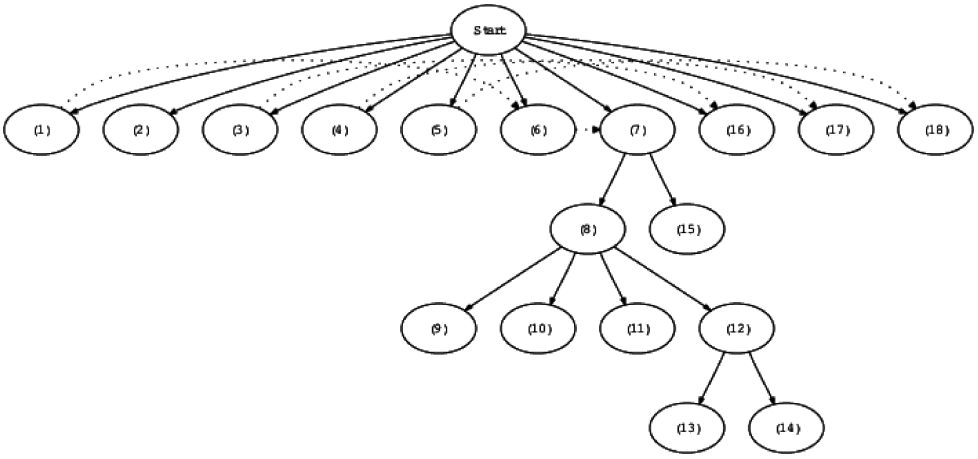


Fig. 15. PDG_{DS} of the program in Figure 1 (a) constructed with respect to the input data $\{text = \text{“}\backslash eof\text{”}, n = 0\}$.

(10)→(17), because statements (9) and (10) are never executed. Therefore, if we use the slicing criterion $\langle 16, \{lines\}, \{text = \text{“}hello\ world!\backslash eof\text{”}, n = 4\}\rangle$, we get the dependence-cache slice of Figure 4. Note that, in this case, deleting those edges from the PDG is enough to get the same precision as a dynamic slicer.

Let us now consider, for the same program an extreme example where the input data is the empty text, that is, $\{text = \text{“}\backslash eof\text{”}, n = 0\}$. The PDG_{DS} for this example is shown in Figure 15. An algorithm to compute such a PDG can be found in Takada et al. [2002].

An experimental comparison by Takada et al. [2002] has shown that dependence-cache slices are, on average, smaller than the corresponding call-mark slices. The same experiments also show that dependence-cache slices are less expensive to be computed.

Question answered. For this particular execution, what program statements can influence these variables at this statement?

Main applications. Debugging, program comprehension, and testing.

2.14. Simultaneous Slicing [Hall 1995]

There are two different views of simultaneous slicing. On the one hand, the first definition is *simultaneous dynamic slicing*—this technique has been also called *union slicing* [Beszédes et al. 2002] in the literature—where a slice is computed with respect to a set of inputs for the program, and thus, the slice can be computed from the dynamic slices computed for each input. However, the construction of such a slice does not simply reduce to the union of slices (this is not sound) and it requires the use of more elaborate methods, such as the *simultaneous dynamic slice* (SDS) procedure [Hall 1995]. On the other hand, Danicic and Harman [1996] define simultaneous slicing as a generalization of program slicing in which a set of slicing criteria is considered.[3] Hence, slices are computed with respect to a set of different points, rather than a set of inputs.

2.14.1. Simultaneous Dynamic Slicing [Hall 1995]. Similarly to quasi-static slicing, simultaneous dynamic slicing computes a slice with respect to a particular set of executions; however, while quasi-static slicing fixes a set of the program inputs (leaving the rest unknown), in simultaneous dynamic slicing [Hall 1995], a set of complete inputs is

[3]Weiser probably already noticed that his algorithm could work with multiple points. However, until 1996, this idea was not exploited.

known. Thus, a simultaneous dynamic slicing criterion can be seen as a set of dynamic slicing criteria.

A simultaneous dynamic slicing criterion has the form $\langle s^i, v, \{I_1, \ldots, I_n\}\rangle$, where i is the number of occurrences of statement s in the execution history, v is the set of variables of interest, and $\{I_1, \ldots, I_n\}$ is a set of complete inputs for the program. For instance, a slice of Figure 1(a) with respect to the slicing criterion $\langle 16^{92}, \{lines\}, \{I_1, I_2\}\rangle$, where $I_1 = (text = \text{“}hello\ \ world!\backslash eof\text{”}, n = 4)$ and $I_2 = (text = \text{“}hello\backslash nworld!\backslash n\ \backslash eof\text{”}, n = 0)$ is depicted in Figure 1(b).

Note that, in contrast to quasi-static slicing where the set of considered executions can be infinite, in simultaneous dynamic slicing it is always finite.

Question answered. For these particular executions, what program statements can influence these variables at this statement?

Main applications. Program reuse, program redesign, and program maintenance.

2.14.2. Simultaneous (Static) Slicing [Danicic and Harman 1996]. Although Weiser's algorithm is able to produce slices starting at many slicing criteria points, the first to talk about simultaneous slicing from a static point of view were Danicic and Harman.

In Danicic and Harman's definition of simultaneous slicing, a slicing criterion is $\langle\{(s_1, v_1), \ldots, (s_n, v_n)\}\rangle$, where $(s_i, v_i), 1 \leq i \leq n$ are static slicing criteria. Therefore, clearly simultaneous slicing is a generalization of static slicing, where a set of points can be considered instead of only one. Following our running example, a simultaneous slice with respect to the slicing criterion $\langle\{(9, lines), (16, lines)\}\rangle$ is shown in Figure 1(b).

This slice has been computed by the union of the static slices for $\langle 9, lines\rangle$ and $\langle 16, lines\rangle$. However, although the union of static slices produces a correct slice in practice, De Lucia et al. [2003] showed that in theory—if we rigidly fit to the definition of slice—"unions of slices are not slices." A sample of this theoretic phenomenon can be found in Figures 1 and 2 of De Lucia et al. [2003].

It should be clear that although the work by Danicic and Harman focused on static slicing, their definition of simultaneous slicing can be easily adapted to other forms of slicing (see Section 3).

Lakhotia [1993] introduced a new kind of slicing criterion in order to compute module cohesion. The aim of his work was to collect the module's components which contribute to the final value of the output variables of this module. Therefore, in this scheme, the slicing criterion is formed by a set of variables. It is called *end slicing* because the slicing point is the end of the program. Then, an end slicing criterion is formed by a set of variables of the program, and thus it can be represented as a set of slicing criteria (i.e., it is a particular case of simultaneous static slicing).

Question answered. What program statements can influence these variables at these statements?

Main applications. Program comprehension, debugging and module cohesion analysis.

2.15. Interface Slicing [Beck and Eichmann 1993]

Interface slicing [Beck 1993; Beck and Eichmann 1993] is a slicing technique which is applied to a module in order to extract a subset of the module's functionality. A module can contain many functions and procedures (in the following, components) that can be used by the system which imports this module. The collection of component names form the interface of the module.

When a programmer imports the module, usually only a part of it is used. The rest of the unused components become dead code. Therefore, the basic idea of interface slicing

```
(1) module operations                    (15) procedure increment(x)
                                         (16) sum(x,1);
(2)  procedure line-char-count()         (17) return
(3)  read(text);
(4)  lines = 1;                          (18) procedure sum(a, b)
(5)  chars = 1;                          (19) a = a + b;
(6)  c = getChar(text);                  (20) return
(7)  while (c != '\eof')
(8)       if (c == '\n')                 (21) procedure decrement(x)
(9)       then sum(lines,1);             (22) rem(x,1);
(10)      else increment(chars);         (23) return
(11)      c = getChar(text);
(12) write(lines);                       (24) procedure rem(a, b)
(13) write(chars);                       (25) a = a - b;
(14) return                              (26) return
```

Fig. 16. Module which exports five procedures.

is to allow the programmer to produce a new module which only contains the desired components. To do this, the programmer only has to specify which part of the interface (i.e., which module components) are at interest. Then, an interface slicer produces a new module from the original module which only contains the desired components.

An interface slicing criterion has the form $\langle f \rangle$, where f is a set of function or procedure names of the module's interface. For instance, consider the module of Figure 16 (whose interface is [$line-char-count$, $increment$, sum, $decrement$ and rem]) and the slicing criterion $\langle \{rem, line-char-count\} \rangle$. The slice produced (the new module) would be formed by line (1) and procedures rem, $line-char-count$, $increment$, and sum. $increment$ and sum are included in the slice because they are referenced by $line-char-count$.

Hence, a component of the module can belong to the slice because it is part of the desired interface, or because it is used (maybe transitively) by a desired part of the interface. Therefore, the process of interface slicing is essentially the same as conventional slicing, but the interesting dependences are defined between components and global variables rather than between statements. Hence, it is the same problem but with a bigger granularity level.

It is important to note that each component of the slicing criterion specifies a point in the module. In principle, interface slicing could be thought as a particular instance of simultaneous static slicing where multiple points are implicitly specified in the slicing criterion. In particular, each component name f could be converted to a static slicing criterion $\langle s, v \rangle$, where s is the last statement of procedure f, and v contains all the variables appearing in f. However, the precision of both methods is different.

While a static slice taken from $\langle s, v \rangle$ would remove dead code appearing in f, an interface slice produced from $\langle f \rangle$ would keep the dead code in f, because the interface slice extracts all the statements in relevant components. Hence, an interface slice is a superset of a simultaneous static slice taken from equivalent slicing criteria.

Question answered. What parts of this module are I needed to reuse these procedures?

Main applications. Reverse engineering and code reuse.

2.16. Program Dicing [Lyle and Weiser 1987]

The process of program dicing [Lyle and Weiser 1987] was originally designed to "remove those statements of a static slice of a variable that appears to be correctly computed from the static slice of an incorrectly valued variable." From this definition, it is easy to deduce that program dicing was originally defined for debugging. In essence,

```
(3)  lines = 1;
(7)  while (c != '\eof')
(8)       if (c == '\n')
(9)       then lines = lines + 1;
(16) write(lines);
```

Fig. 17. Stop-list slice of the program in Figure 1 with respect to the criterion $\langle 16, \{lines\}, \{c\}\rangle$.

a dice is the set difference between at least two slices—typically, one of an incorrect value and the other of a correct value. The main idea is that after some tests, we can find some correct and incorrect outputs. The statements in a slice of an incorrect output that do not belong to the slice of a correct output are more likely to be wrong. As a consequence, a program dicing criterion specifies n points in the program, for example, one for the correct computation and $n - 1$ for the wrong computations. For instance, a backward static program dicing criterion for the program in Figure 1(a) could be $\langle(16, \{lines\}), \{(17, \{chars\})\}\rangle$; its meaning is that variable *lines* at line (16) produced an incorrect value (its slice contains the statements 1, 3, 6, 7, 8, 9, 15, and 16), whereas variable *chars* at line 17 produced a correct value (its slice contains the statements 1, 4, 6, 7, 8, 10, 11, 15, and 17). Note that only one point has been specified for the correct computation, but a set is possible. The dice computed with respect to this criterion is shown in Figure 5.

In contrast to program slicing where a dynamic slice is included in its corresponding static slice, a dynamic dice is not necessarily included in its corresponding static dice. Chen and Cheung [1993] investigated under what conditions a dynamic dice is more precise than a static dice.

Question answered. What program statements can influence these variables at this statement but do not influence these other variables?

Main applications. Debugging.

2.17. Stop-List Slicing [Gallagher et al. 2006]

Stop-list slicing [Gallagher et al. 2006] is a slicing technique similar to dicing, because both of them use variables of no interest to the programmer to reduce the size of the slice. Usually, programs contain two kinds of variables: those which perform computations (e.g., output variables) and those which help to perform computations (e.g., auxiliary variables such as temporaries, counters, and indices). Clearly, the importance of these variables is different depending on the purpose of the programmer. Hence, the objective of stop-list slicing is to allow the programmer to remove those variables from the slice which are of no interest.

Stop-list slicing augments the slicing criterion with a new list of variables: the set of variables considered uninteresting. Therefore, a stop-list slicing criterion has the form $\langle s, v, v_{sl}\rangle$, where s and v have the same meaning as in static slicing, and v_{sl} is the stop-list variable set—the variables of no interest. The stop-list variable set is used to purge the dependence graph by removing all the simple assignments to the variables in the stop-list set and all the data dependencies starting from them. Note that control dependencies are not removed.

After the dependence graph is purged, the slice is computed as usual with the standard graph reachability algorithm. Clearly, since some parts of the dependence graph are missing, the slice produced is smaller.

As an example, a stop-list slicing criterion for our running example could be $\langle 16, \{lines\}, \{c\}\rangle$, which denotes that the computation of variable c is uninteresting. The slice produced for this criterion is shown in Figure 17.

Question answered. What program statements can influence these variables at this statement (but I am not interested in the statements needed for the computation of the value of this set of other variables)?

Main applications. Program comprehension and debugging.

2.18. Barrier Slicing [Krinke 2003]

Barrier slicing was introduced by Jens Krinke [2003, 2004] as a novel form of slicing in which the programmer has more control over the construction of the slice. In particular, in this technique, the programmer can specify which parts of the program can be traversed when constructing the slice and which parts cannot. This can be useful in debugging. For instance, programmers often reuse code which is known to be correct. When debugging, the programmer might want to exclude this code from the slice (because it cannot contain the bug which is being looking for).

This ability can be used by including *barriers* in the slicing criterion. A barrier is specified with a set of nodes (or edges) of the PDG that cannot be passed during the graph traversal. Therefore, a barrier slice can be computed by stopping the computation of the transitive closure of the program dependencies whenever a barrier is reached. A usual barrier-slicing criterion is a tuple $\langle s, v, b\rangle$, where s and v have the same meaning as in static slicing, and b is a collection of statement numbers denoting the barriers. For instance, in our running example, we could use the slicing criterion $\langle(16, \{lines\}), \{8\}\rangle$, which can be interpreted as from those executions that do not execute the if-then-else, what statements do influence variable *lines* at line 16? The slice produced for this criterion is depicted in Figure 4.

Question answered. What program statements can influence these variables at this statement from this set of other statements?

Main applications. Program comprehension, remote software trusting, and debugging.

2.19. Conditioned Slicing [Ning et al. 1994]

Although Ning et al. [1994] were the first to work with conditioned slices; this technique was formally defined for the first time by Canfora et al. [1994].

Similarly to simultaneous dynamic slicing and quasi-static slicing, conditioned slicing [Canfora et al. 1994, 1998] computes slices with respect to a set of initial states of the program. The original definition proposed the use of a condition (from the programming language notation) to specify the set of initial states. Posteriorly, Field et al. [1995] proposed the same idea (known as *parametric program slicing* or *constraint slicing*) based on constraints over the initial values. Finally, De Lucia et al. [1996] proposed the condition to be a universally quantified formula of first-order predicate logic. In this approach, which is a generalization of the previous works, a conditioned slicing criterion is a quadruple $\langle i, F, s, v\rangle$, where i is a subset of the input variables of the program, F is a logic formula on i, s is a statement of the program, and v is a subset of the variables in the program.

The logic formula F identifies a set of the possible inputs of the program, which could be infinite. For instance, consider the conditioned slicing criterion $\langle(text, n), F, 18, \{subtext\}\rangle$, where $F = (\forall c \in text,\ c \neq \text{`}\backslash n\text{'} \ .\ n > 0)$; the conditioned slice of the program in Figure 1(a) with respect to this criterion is shown in Figure 18.

It should be clear that conditioned slicing is a generalization of both simultaneous dynamic slicing and quasi-static slicing because their respective slicing criteria are a particular case of a conditioned slicing criterion. As an advantage, conditioned slicing

```
(1)  read(text);
(2)  read(n);
(5)  subtext = "";
(6)  c = getChar(text);
(7)  while (c != '\eof')
(8)       if (c == '\n')
(12)           if (n != 0)
(13)           then subtext = subtext ++ c;
(14)                n = n - 1;
(15)       c = getChar(text);
(18) write(subtext);
```

Fig. 18. Conditioned slice of Figure 1(a) with respect to $\langle(text, n), F, 18, \{subtext\}\rangle$, where $F = (\forall c \in text,\ c \neq$ '$\backslash n$' $.\ n > 0)$.

allows us to specify relations between input values. For instance, condition F specifies that if `text` is a single line, then `n` must be higher than 0.

It is important to note that the claim "conditioned slicing generalizes dynamic slicing" is imprecise. To be more precise, the correct sentence should be "path-aware conditioned slicing generalizes path-aware dynamic slicing" or "path-unaware conditioned slicing generalizes path-unaware dynamic slicing." But the sentence "path-unaware conditioned slicing generalizes path-aware dynamic slicing" is not true (see Section 2.3). Therefore, in the following, whenever we say that technique A is a generalization of technique B, we will implicitly assume that both techniques use equal non-mentioned conditions (i.e., both are static or dynamic, both are forward or backward, both are path-aware or path-unaware, etc.). Section 3 studies these conditions and its possible values to make two slicing techniques comparable.

For instance, following our running example, the conditioned slicing criterion $\langle(text), F, 16, \{lines\}\rangle$ where $F = (\forall c \in text, c \neq$ '$\backslash n$') generalizes the dynamic slicing criterion of Figure 4. Here, the inputs considered are all those texts with a single line—an infinite set—and thus, the corresponding slice is shown in Figure 4.

Question answered. For the initial states which satisfy this condition, what statements can influence these variables at this statement?

Main applications. Debugging, software reuse, ripple effect analysis, legacy code understanding, and program comprehension.

2.20. Backward Conditioning Slicing [Fox et al. 2001]

The definition of conditioned slicing defined by De Lucia et al. [1996] has been later referred to as *forward conditioning slicing*, because the specified condition affects the initial state—the input, and this condition is used forward to determine which statements are executed when the condition is true. Therefore, forward conditioning is able to answer questions of the form "what happens when the initial state is s?"

Fox et al. [2001] noted that the condition could be placed at any point on the program. This approach is called *backward conditioning slicing*, because the specified condition is used backwards in the program to determine which statements are needed to make the condition true. Therefore, backward conditioning is able to answer questions of the form "how could the program get into state *s*?" As happens with conventional forward slicing, backward conditioning slicing requires symbolic execution and theorem-proving mechanisms.

Together with the definition of backward conditioning slicing, Fox et al. introduced their generalization to consider a set of conditions instead of one. As a result, the

```
(3)  lines = 1;
(16) write(lines);
```

Fig. 19. Backward conditioning slice of Figure 1(a) with respect to $\langle\{(\{lines\}, 16), (lines = 1, 16)\}\rangle$.

backward conditioning slicing criterion is defined as a set of pairs where each pair is either (1) a set of variables and a program point (a traditional static slicing criterion), or (2) a condition and a program point.

In our running example, we could produce the slice shown in Figure 19 from the slicing criterion $\langle\{(\{lines\}, 16), (lines = 1, 16)\}\rangle$.

As a program comprehension tool, backward conditioning slicing can be used to check that some situations are not possible in our program. For instance, the backward conditioning slice of Figure 1(a) with respect to the backward conditioning slicing criterion $\langle\{(\{lines\}, 18), (lines = 0, 18)\}\rangle$ would be empty, denoting that it is not possible to reach the state in which (lines = 0,18).

Question answered. What statements can influence these variables at this statement when these conditions at these (other) statements are satisfied?

Main applications. Program specialization and program comprehension.

2.21. Pre/Postconditioned Slicing [Harman et al. 2001]

Pre/postconditioned slicing [Harman et al. 2001] is a generalized form of conditioned slicing which combines forward and backward conditioning; therefore, it simultaneously uses both the forward conditions (called preconditions) of forward conditioning slicing and the backward conditions (called postconditions when negated) of backward conditioning slicing. In pre/postconditioned slicing, slices are constructed by removing all the statements except those which are in the execution paths determined by the precondition and which can lead to the satisfaction of the negation of the postcondition. The reason for negating the postcondition is the verification that the program always satisfies it. For instance, if we want to check that a program executed in an initial state satisfying precondition C_1 always ends in a final state satisfying postcondition C_2, we would produce a slice with respect to the forward condition C_1 and the backward condition $\neg C_2$. Then, the slice should be empty. If it is not, the statements in the slice are those which could lead to the violation of the postcondition.

From the previous discussion, it is easy to deduce that a pre/postconditioned slicing criterion can use forward and backward conditions at the same time; moreover, these conditions can be more than two if we allow conditions to be inserted at arbitrary program points. Therefore, to distinguish between forward and backward conditions, a special arrow notation is used [Fox et al. 2001], where $\downarrow \lceil c \rceil$ denotes a forward condition c, and $\uparrow \lceil c \rceil$ denotes a backward condition c.

A pre/postconditioned slicing criterion is equal to a backward conditioning slicing criterion except that each condition pair in the criterion is augmented with the arrow notation. In our running example we could use the criterion $\{(\{lines\}, 16), (\downarrow \lceil$'\n'$\in text\rceil, 1), (\uparrow \lceil lines <= 1\rceil, 18)\}$ which would produce the empty slice to denote that whenever variable *text* contains a '\n' in the initial state, variable *lines* must be greater than 1 in the final state.

Question answered. For those executions whose inputs satisfy this condition, what statements can influence these variables at this statement when these other conditions at these (other) statements are satisfied?

Main applications. Program comprehension, reuse, and verification.

```
(1)  read(text);
(6)  c = getChar(text);
(7)  while (c != '\eof')
(8)       if (c == '\n')
(9)       then lines = lines + 1;
```

Fig. 20. Path slice of the program in Figure 1(a) with respect to $\langle 9, \{1, 2, 3, 4, 5, 6, 7, 8, 9\}\rangle$.

2.22. Path Slicing [Jhala and Majumdar 2005]

Path slicing [Jhala and Majumdar 2005] is a program slicing-based technique that tries to answer the following questions.

(1) Given an execution path, which statements can possibly influence reachability of a given statement s?
(2) Why is this path never executed?
(3) Why doesn't my program ever execute statement s in this path?

From these questions, it is easy to realize that path slicing is computed with respect to a path in the CFG. It should be clear that a path in the CFG corresponds to a set of possible infinite inputs. Therefore, path slicing is different from static slicing where all possible executions are considered, and it is different from dynamic slicing where only one execution is considered. Moreover, path slicing is different from quasi-static slicing, simultaneous dynamic slicing, hybrid slicing, and pre/postconditioned slicing, because all of them consider feasible computations. In contrast, path slicing can handle infeasible computations.

A path slicing criterion is a pair $\langle s, p\rangle$, where s is the statement of interest, and $p = \{s_1, \ldots, s_n\}$ is a sequence of statements defining a possibly infeasible subpath in the CFG of the program. The path slice is then computed by removing from p those statements that cannot influence the reachability of s.

For instance, a path slice of the program in Figure 1(a) with respect to the path slicing criterion $\langle 9, \{1, 2, 3, 4, 5, 6, 7, 8, 9\}\rangle$ is shown in Figure 20.

Statements (7) and (8) are needed to execute (9) (due to control dependence). Statement (6) is needed to execute (7), and (1) to execute (6) (due to data dependence).

The definition of a path slice is the following Jhala and Majumdar [2005]: A slice of a path π is a subsequence of the edges of π such that (1) (complete) whenever the sequence of operations labeling the subsequence is feasible, the target location is reachable modulo termination, and (2) (sound) whenever the sequence of operations labeling the subsequence is infeasible, the path is infeasible.

This means that one limitation of path slicing is that it avoids the difficult question of statically reasoning about termination. Therefore, researchers that work with nonterminating programs should take into account that the feasibility of a path slice guarantees that either the target location is reachable or that all states that can execute the path slice cause the program to enter an infinite loop.

The interesting part of path slicing is that it can slice infeasible paths and thus reason about questions like why a path is never executed? For instance, consider the program in Figure 21(a) where the statements of function `bigFunction` are skipped for simplicity.

In this program, statement (7) is never executed because if $x > 0$ holds in statement (2), then variable y is set to 1; hence, y cannot be 0 at statement (6). Therefore, the path $\{1, 2, 3, 4, bigFunction, 5, 6, 7\}$ is infeasible. However, path slicing allows us to produce a slice with respect to this path. This can be useful in debugging to reason why this path is infeasible when it should be. It also allows us to reason about why statement (7) is not executed in this path. As an example, Figure 21(b) shows the path slice of the

```
(1)  read(x);                    (1)  read(x);
(2)  if (x>0)                    (2)  if (x>0)
(3)     y=1;                     (3)     y=1;
(4)  z = bigFunction(y);
(5)  if (x>0)                    (5)  if (x>0)
(6)     if (y=0)                 (6)     if (y=0)
(7)        write("ERROR")        (7)        write("ERROR")
```

(a) Example program. (b) Path slice.

Fig. 21. Example program (a) and its path slice (b) with respect to $\langle 7, \{1, 2, 3, 4, 5, 6, 7\}\rangle$.

program in Figure 21(a) with respect to the criterion $\langle 7, \{1, 2, 3, 4, 5, 6, 7\}\rangle$. This slice shows that the code in function *bigFunction* can not influence execution of statement (7).

In our running example, the path slicing criterion $\langle 16, \{1, 2, 3, 4, 5, 6, 7, 16, 17, 18\}\rangle$ yields the empty slice, because none of the previous statements can avoid the execution of statement (16).

Question answered. Given this execution path, which statements can possibly influence reachability of this statement?

Main applications. Debugging and testing.

2.23. Abstract Slicing [Hong et al. 2005]

Static slicing is able to determine for each statement in a program whether it affects or is affected by the slicing criterion; however, it is not able to determine under which variable values the statements affect or are affected by the slicing criterion. This problem is overcome by abstract slicing [Hong et al. 2005].

Essentially, abstract slicing extends static slicing with predicates and constraints that are processed with an abstract interpretation and model checking-based technique. Given a predicate of the program, every statement is labeled with the value of this predicate that makes the statement affect (or be affected by) the slicing criterion. Similarly, given a constraint on some variables of the program, every statement is labeled indicating if they can or cannot satisfy the constraint. Clearly, static slicing is a particular instance of abstract slicing, where neither predicates nor constraints are used. Moreover, in a way similar to conditioned slicing, abstract slicing is able to produce slices with respect to a set of executions by restricting the allowed inputs of the program using constraints.

In abstract slicing, a slicing criterion has the form $\langle s, P, C\rangle$, where s is a statement of the program, and P and C are, respectively, a predicate and a constraint for some statements defined over some variables of the program. For instance, C could be $((1), y > x)$, meaning that at statement (1), the condition $y > x$ holds. Based on previous slicing techniques, Hong et al. adapted this technique to forward/backward slicing and chopping, giving rise to abstract forward/backward slicing and abstract chopping.

As an example, consider the program in Figure 22(a); the abstract slice of this program with respect to the slicing criterion $\langle (6), y > x, True\rangle$ is depicted in Figure 22(b). Here, we are interested in knowing the condition under each statement that affects the slicing criterion; we do not impose any condition. In the slice, we see that the statement labeled with '[*true*]' will always affect the slicing criterion; '$max = x$' will only affect the slicing criterion if $y \leq x$; and the 'if-then' statements will affect the slicing criterion if $y > x$.

In our running example, we can produce a slice with respect to the slicing criterion $\langle 16, c =$'\n', [(1), *text* = "*hello world*!*eof*"]$\rangle$. This criterion uses the same dynamic input

```
(1)  read(x);                    (1)  read(x);        [true]
(2)  read(y);                    (2)  read(y);        [true]
(3)  max = x;                    (3)  max = x;        [not(y>x)]
(4)  if (y > x)                  (4)  if (y > x)      [y>x]
(5)  then max = y;               (5)  then max = y;   [y>x]
(6)  write(max);                 (6)  write(max);     [true]

   (a) Example program.             (b) Abstract slice.
```

Fig. 22. Abstract slicing: (b) is an abstract slice of (a) with respect to $\langle(6), y > x, True\rangle$.

```
(1)  read(text);                                  [false]
(2)  read(n);                                     [false]
(3)  lines = 1;                                   [c!='\n']
(4)  chars = 1;                                   [false]
(5)  subtext = "";                                [false]
(6)  c = getChar(text);                           [false]
(7)  while (c != '\eof')                          [false]
(8)       if (c == '\n')                          [false]
(9)       then lines = lines + 1;                 [false]
(10)           chars = chars + 1;                 [false]
(11)      else chars = chars + 1;                 [false]
(12)           if (n != 0)                        [false]
(13)           then subtext = subtext ++ c;       [false]
(14)                n = n - 1;                    [false]
(15)      c = getChar(text);                      [false]
(16) write(lines);                                [c!='\n']
(17) write(chars);                                [false]
(18) write(subtext);                              [false]
```

Fig. 23. Abstract slice of Figure 1(a) with respect to $\langle 16, c =$'\n', [(1), $text =$ "*hello world*!*eof*"]$\rangle$.

as in previous criteria, thanks to the constraint used. Therefore, with this input, the slice produced would be similar to the one shown in Figure 4 (a dynamic slice). This can be seen in Figure 23 by looking at the conditions on the right. All the statements labeled with `false` cannot affect statement (16). Moreover, the abstract slice provides additional information: it says, for each statement in the slice, whether the condition $c =$'\n' will be satisfied or not.

Question answered. Under which variable values do the program's statements affect or are affected by the slicing criterion?

Main applications. Program comprehension.

2.24. Amorphous Slicing [Harman and Danicic 1997]

All approaches to slicing discussed so far have been based on two assumptions: the slice preserves (part of) the semantics of the program, and it is *syntax preserving*, that is, the slice is a subset of the original program statements. In contrast, amorphous slices [Harman and Danicic 1997] preserve the semantics restriction, but they drop the syntactic restriction: amorphous slices are constructed using some program transformation which simplifies the program and preserves the semantics of the program with respect to the slicing criterion. In the literature (see, e.g., [Ward 2002, 2003; Ward and Zedan 2007], when a slicing technique is restricted to statement deletion, it is also referred to as *syntactic slicing* as opposed to *semantic slicing*, where only the semantic restriction must be preserved.

```
(1) program main
(2) chars = sum(chars,1);
(3) end

(4) procedure increment(a)
(5) sum(a,1);
(6) return

(7) procedure sum(a,b)
(8) a = a + b;
(9) return
```

Program 1

```
(1) program main
(2) chars = sum(chars,1);
(3) end

(7) procedure sum(a,b)
(8) a = a + b;
(9) return
```

Program 2

```
(1) program main
(2) chars = chars + 1;
(3) end
```

Program 3

Fig. 24. Example of amorphous slicing.

The syntactic freedom allows amorphous slicing to perform greater simplifications, thus often being considerably smaller than conventional program slicing. These simplifications are very convenient in the context of program comprehension where the user needs to simplify the program as much as possible in order to understand a part of the semantics of the program having syntax at less importance.

For instance, consider `Program 1` in Figure 24 together with the slicing criterion $\langle 3, \{chars\}\rangle$. An intraprocedural static slice of this program would contain the whole program (`Program 1`). In contrast, an interprocedural static slice would remove statements (4), (5), and (6) (`Program 2`). Finally, its amorphous slice would be `Program 3`. It should be clear that the three programs preserve the same semantics, but `Program 3` has been further simplified by partially evaluating [Jones 1996] some expressions (thus changing the syntax) of `Program 1`. Clearly, `Program 3` is much more understandable than `Program 1` and `Program 2`.

It is known [Harman and Danicic 1997] that amorphous static slicing subsumes traditional static slicing, that is, there will always be an amorphous static slice which is at least as thin as the static slice constructed for the same slicing criterion. In addition, there will usually be an amorphous slice which is thinner than the associated static slice. This leads to the search for the minimal amorphous slice. However, as proved by Harman and Danicic [1997], the computation of the minimal amorphous static slice of an arbitrary program is, in general, undecidable.

Amorphous slicing generalizes static slicing with respect to the syntax preserving dimension. While static slicing is restricted to one syntax (that of the original program), amorphous slicing can produce slices with different syntax modifications. This new dimension can be combined with other techniques besides static slicing. For instance, *conditioned amorphous slicing* introduces the non-syntax preserving feature of amorphous slicing into the conditioned slicing technique (see Section 2.19). To continue with our running example, Figure 25 shows a conditioned amorphous slice of the program in Figure 9 with respect to the slicing criterion $\langle (text), (text = \text{“}hello\backslash nworld!\backslash n\backslash eof\text{”}), 13, \{lines\}\rangle$.

Question answered. Can this program be changed to only compute these variables at this statement?

Main applications. Program comprehension.

2.25. Decomposition Slicing [Gallagher and Lyle 1991]

Decomposition slicing [Gallagher and Lyle 1991] was introduced in the context of software maintenance to capture all computation on a given variable. The objective of

```
(1)  program main
(2)  read(text);
(3)  lines = 1;
(5)  c = getChar(text);
(6)  while (c != '\eof')
(7)       if (c == '\n')
(8)       then lines = lines +1;
(10)      c = getChar(text);
(11) write(lines);
```

Fig. 25. Conditioned amorphous slice of the program in Figure 9 with respect to the slicing criterion $\langle(text), (text = \text{“}hello\backslash nworld!\backslash n\ \backslash eof\text{”}), 13, \{lines\}\rangle$.

this technique is to extract those program statements which are needed to compute the values of a given variable. Therefore, a decomposition slicing criterion is composed of a single variable v. The slice is then built from the union of the static backward slices constructed for the criteria $\{\langle n_1, v\rangle, \ldots, \langle n_m, v\rangle, \langle end, v\rangle\}$ where $\{n_1, \ldots, n_m\}$ is the set of lines in which v is output and *end* is the end of the program. It should be clear that decomposition slicing is an instance of simultaneous slicing, where the set of slicing criteria is derived from the variable and the program.

A decomposition slice of the program in Example 1(a) with respect to the slicing criterion $\langle lines\rangle$ is shown in Figure 1(b).

Question answered. What statements of the program can affect this variable?

Main applications. Software maintenance, testing, and program comprehension.

2.26. Concurrent Slicing [Cheng 1993]

Since Cheng [1993] defined the first approach, there have been many works [Krinke 1998; Nanda and Ramesh 2000; Müller-Olm and Seidl 2001; Krinke 2003b; Giffhorn and Hammer 2007] that face the complexity introduced by concurrent programs in program slicing.

Concurrent programs cannot be represented with standard graph representations such as the PDG or the SDG, because they allow that some parts of the program are executed in parallel. These pieces of code that can be executed in parallel are called threads.

In order to represent threads, the CFG and the PDG are extended with special nodes that represent the parallel execution of threads. These extensions are called, respectively, threaded CFG (tCFG) and threaded PDG (tPDG).

Threads introduce an additional complexity to program slicing when they can be synchronized or can communicate (e.g., through the use of variables), because they introduce a new kind of dependence (called interference) between statements.

A statement s_1 is interference dependent on a statement s_2 if the following holds.

—s_2 defines a variable which is used in s_1 and
—s_1 and s_2 may be potentially executed in parallel.

The main problem of interference is that it is not transitive, as control and data dependence are. Therefore, slicing algorithms for concurrent programs (see, e.g., [Nanda and Ramesh 2000; Krinke 2003b]) must provide a special treatment for interference. An evaluation of slicing algorithms for concurrent programs is presented in Giffhorn and Hammer [2007].

Question answered. What program statements can influence these variables at this statement in this concurrent program?

Main applications. Debugging and program comprehension.

```
(1)  program main
(2)  read(x);
(3)  y = 0;
(4)  z = 1;
(5)  if (x > 5)
(6)  then p = &y;
(7)  else p = &z;
(8)  *p = *p + 1;

      Program 1
```

```
(1)  program main
(2)  read(x);
(5)  if (x > 5)
(6)  then p = &y;
(7)  else p = &z;
(8)  *p = *p + 1;

      Program 2
```

```
(1)  program main
(3)  y = 0;
(8)  *p = *p + 1;

      Program 3
```

Fig. 26. Example program with a pointer p and its incremental slices with respect to the slicing criteria $\langle 8, y, def\_use \rangle$ (program 2) and $\langle 8, y, pos\_use \rangle$ (program 3).

2.27. Incremental Slicing [Orso et al. 2001a, 2001b]

Incremental slicing [Orso et al. 2001b] is based on the idea that not all data dependencies are equal. In particular, Orso et al. [2001a] distinguish between 24 different types of data dependencies. Their classification is based on the different levels of complexity that can be introduced by pointers in a program. For instance, consider Program 1 in Figure 26.

Here, variable x is only defined at statement (2), and it is used for the first time at statement (5). Therefore, x at statement (5) data depends on x at statement (2). Note that in all executions, the value of x at statement (5) will be the same as the value of x at statement (2). In contrast, the definition in statement (8) can modify either y or z depending on how the predicate in statement (5) evaluates. Hence, these definitions can be classified differently: the definition of y (or z) at statement (8) is a possible definition whereas the definition of x at statement (2) is a definite definition. Similarly, uses also can be classified as possible or definite. The combination of these types of definitions, uses, and possible paths where they could appear, gives rise to 24 different types of data dependencies.

The objective of this technique is to allow the user to focus on a particular type of data dependence. This can be useful, for example, for program comprehension, where the user can initially ignore weak data dependencies and concentrate on strong data dependencies. Then, weaker data dependences in the slice can be incrementally incorporated. This approach allows us to produce smaller and, thus, easier to understand slices. Alternatively, incremental slicing can be used in debugging by using the opposite strategy. The programmer can produce slices by only considering weak data dependences which are less obvious and, thus, more likely to be buggy.

An incremental slicing criterion is a triple $\langle s, v, t \rangle$, which specifies a statement s, a set of variables v, and a set of types of data dependences t. Hence, an incremental slice contains those statements that may affect, or may be affected by, the values of the variables in v at s through transitive control or specified types of data dependencies.

As an example, the slice of Program 1 in Figure 26 with respect to the slicing criterion $\langle 8, y, all \rangle$, where *all* stands for all types of data dependencies, is the whole Program 1. The slice of Program 1 with respect to the slicing criterion $\langle 8, y, def\_use \rangle$, where *def_use* stands for all types of data dependencies with a definite use of the variable involved, is Program 2. Finally, the slice of Program 1 with respect to the slicing criterion $\langle 8, y, pos\_use \rangle$, where *pos_use* stands for all types of data dependencies with a possible use of the variable involved, is Program 3.

Because there are no pointers in the program in Example 1(a), only definite definitions and uses exist. An incremental slice of this program with respect to the slicing criterion $\langle 16, lines, def\_use \rangle$ is shown in Figure 1(b).

```
(1)  x = new A();
(2)  y = new B();
(3)  z = x;
(4)  w = x;
(5)  w.f = y;
(6)  if (z == w)
(7)  then v = z.f;
```

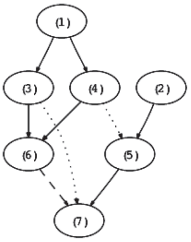
Program 1

```
(2)  y = new B();

(5)  w.f = y;

(7)  then v = z.f;
```

Program 2



PDG of Program 1

Fig. 27. Example of a thin slice.

Clearly, incremental slicing generalizes static slicing by only using a subset of data dependencies to compute the slice. The same idea has been later used in *thin slicing* [Sridharan et al. 2007]. Thin slicing distinguishes between three types of dependences.

(1) *Control dependencies*.
(2) *Base pointer flow dependencies*. (A base pointer flow dependence is a flow dependence due solely to the use of a pointer in a field dereference).
(3) *Producer flow dependencies* (i.e., those flow dependencies which are not base pointer flow dependencies).

A thin slice is computed by ignoring control and base pointer flow dependencies; thus, a thin slice only includes those statements related by a producer flow dependence. These statements are called *producer statements*. Informally, statement s is a producer for statement t if s is part of a chain of assignments that computes and copies a value to t.

As an example, consider the Java code in Figure 27.

In the PDG, thick arrows represent producer flow dependencies, dotted arrows represent base pointer flow dependencies, and dashed arrows represent control dependencies. Therefore, following producer flow dependencies, we see that `Program 2` is a thin slice of `Program 1` with respect to $\langle 7, v\rangle$. Observe that the static slice with respect to the same criterion is the entire `Program 1`.

In our running example, a thin slice of the program in Example 1(a) with respect to the slicing criterion $\langle 16, lines\rangle$ is shown in Figure 5.

Question answered. What program statements can influence these variables at this statement if we only consider these particular types of data dependence?

Main applications. Debugging and program comprehension.

2.28. Proposition-Based Slicing [Dwyer and Hatcliff 1999]

Proposition-based slicing [Dwyer and Hatcliff 1999] was defined to reduce the finite-state transition system used in verification techniques such as model checking. Because properties verification is often a very costly task, this reduction allows the user to cope with bigger and more complex programs.

In proposition-based slicing, the final objective is to perform model checking with respect to a linear temporal logic (LTL) formula. Therefore, the user specifies a formula, and the slicing criterion must be derived from the formula. In particular, given a LTL formula, the slicing criterion produced is a simultaneous static slicing criterion because it contains a set of slicing points.

For instance, given a LTL formula $\psi = \diamond(16) \Rightarrow lines > 0$ (whenever statement (16) is executed, variable *lines* is greater than 0), the following slicing criterion is produced:

$\langle\{(7, lines), (16, lines), (17, lines), (3, lines), (9, lines)\}\rangle$. This slicing criterion includes a pair (s, v), where v is the set of variables in ψ in the following cases.

—For each statement appearing in ψ (in the example, statement (16)).
—For each predecessor of the statements appearing in ψ (in the example, statement (7), see the CFG in Figure 2).
—For each successor of the statements appearing in ψ (in the example, statement (17)).
—For each statement which assigns a value to the variables in ψ (in the example, statements (3) and (9)).

The main peculiarity of proposition-based slicing is that, in contrast to previous techniques, slices produced contain statements that do not contribute to the final value of the variables in the slicing criterion. In particular, all the statements appearing in the slicing criterion are included in the slice, indeed, if they cannot affect the specified variables.

As an example, the proposition-based slice of the program in Example 1(a) with respect to the previous slicing criterion $\langle\{(7, lines), (16, lines), (17, lines), (3, lines), (9, lines)\}\rangle$ is shown in Figure 1(b). This slicing criterion generated from ψ ensures that the slice produced satisfies ψ. A justification can be found in Dwyer and Hatcliff [1999].

Question answered. What subset of program statements is needed to satisfy a given LTL formula?

Main applications. Model checking.

2.29. Database Slicing [Sivagurunathan et al. 1997]

The term database slicing can be used in two contexts. First, it can be used to refer to a slicing technique that correctly accounts for database operations. Second, it can be used to refer to the slicing of databases.

Sivagurunathan et al. [1997] noted that standard algorithms produce incorrect slices in the presence of I/O and database operations. The reason being that program slicers only consider the program state and do not take into account the external (or contextual) state (i.e., the state of the external interacting components such as files, databases, user inputs, etc.).

This can be shown with a simple example.

```
(1)  read(x)                                   (1)  read(x)
(2)  read(y)          (2)  read(y)             (2)  read(y)
(3)  z = y +1         (3)  z = y +1            (3)  z = y +1

  Program P     P'=Incorrect slice of P     P''=Correct slice of P
```

If we assume that the command `read()` reads a value from the input file, and the initial state of the file is "42 5", then the value of `z` at line `(3)` when executing `P` is 6, whereas it is 43 when executing `P'`.

Clearly, the command `read()` has an effect on the external state which slicing algorithms should take into account. This problem is common when handling database operations.

The solution proposed by Sivagurunathan et al. was to use special (artificial) variables in the program associated to I/O operations that make the external state accessible to the slicer. The main problem of this solution is that it is necessary to use a transformation schema that maps the original program language to a new language that includes the special variables. Tan and Ling [1998] proposed a similar solution for

database operations. Their approach assumes the existence of implicit variables which are updated with each database operation.

Later, Willmor et al. [2004] proposed an alternative solution which is based on two new types of data dependencies which must be computed and added to the PDG—the resultant PDG is known as *database-oriented program dependence graph* (DOPDG). The new dependencies are program-database dependencies which relate non-database statements with database statements, and database-database dependencies that capture the situation when execution of one database statement affects the behavior of some other database statement that is executed after it.

As an example, the database slice of the program in Example 1(a) with respect to the slicing criterion $\langle 16, lines \rangle$ is shown in Figure 1(b).

Database slicing also refers to the slicing of databases. Cheney [2007] applied program slicing ideas to databases in order to determine what parts of a database may influence the result of a query.

Question answered. What program statements can influence these variables at this statement, taking into account that the program has I/O operations?

Main applications. Program comprehension, debugging, algorithmic debugging, dead code removal, program segmentation, program analysis, software quality assurance, program differencing, software maintenance, testing, program parallelization, module cohesion analysis, partial evaluation, and program integration.

3. DISCUSSION

This section compares all the slicing techniques presented from different points of view. First, the slicing criteria are thoroughly compared by examining their differences for every slicing criterion's dimensions. And second, the techniques are classified with respect to a set of slicing relations, which allows us to produce a hierarchy of slicing techniques.

3.1. A Classification of Slicing Techniques

In this section, we use a classification by Harman et al. [1996] in order to compare and classify all slicing techniques presented so far.

For concreteness, we first need to formally define the notion of *precision* when we talk about precise slices. In general, it is accepted that a slice $\mathcal{S}$ produced for a program $\mathcal{P}$ with respect to a slicing criterion $\mathcal{C}$—for simplicity, we assume here a backward static slicing criterion—is precise if and only if $\mathcal{S}$ contains all and only the statements in $\mathcal{P}$ which can affect $\mathcal{C}$.

However, it is known [Weiser 1984] that this notion of precision is, in general, undecidable. Therefore, we will use a more relaxed (though decidable) notion of precision.

Given a program $\mathcal{P}$, we consider that a backward static slice $\mathcal{S}$ produced for $\mathcal{P}$ with respect to the criterion $\langle s, v \rangle$ is precise if and only if $\forall\, n, n \in \mathcal{S}\ .\ n \rightarrow^* s$. Where $s_1 \rightarrow s_2$ means that s_2 control or data depends on s_1, and $\rightarrow^*$ is the reflexive and transitive closure of $\rightarrow$. For those techniques that use the PDG, this condition is equivalent to saying that there exists a path from n to s in the PDG.

It should be clear that, according to this notion of precision, a precise slice can include statements which cannot influence the slicing criterion. For instance, consider the following program.

```
(1) x = 42;
(2) y = 1 + x;
(3) y = y - x;
```

Table I. Classification of Program Slicing Techniques

Slicing Technique	Vars	Initial States	Slice Points	Sta. Con.	Iter. Cou.	Pa. Aw.	Dir.	Pr.	Lim.	Sy/Se. Pres.	Dep. Con.	Sta. Cons.
Static	$\mathcal{P}(\mathcal{I})$	$\{\mathcal{S}\}$	single	no	$\{\mathcal{N}\}$	no	B	yes	no	$y/y(e)$	$\{\mathcal{D}\}$	$\{\mathcal{T}\}$
KL Dynamic	$\mathcal{P}(\mathcal{I})$	single	single	no	single	yes	B	yes	no	$y/y(e)$	$\{\mathcal{D}\}$	$\{\mathcal{T}\}$
AH Dynamic	$\mathcal{P}(\mathcal{I})$	single	single	no	single	no	B	yes	no	$y/y(e)$	$\{\mathcal{D}\}$	$\{\mathcal{T}\}$
Forward	$\mathcal{P}(\mathcal{I})$	$\{\mathcal{S}\}$	single	no	$\{\mathcal{N}\}$	no	F	yes	no	y/y	$\{\mathcal{D}\}$	$\{\mathcal{T}\}$
Quasi-Static	$\mathcal{P}(\mathcal{I})$	$\mathcal{P}(\mathcal{S})_{\star}$	single	no	$\{\mathcal{N}\}$	no	B	yes	no	$y/y(e)$	$\{\mathcal{D}\}$	$\{\mathcal{T}\}$
Conditioned	$\mathcal{P}(\mathcal{I})$	$\mathcal{P}(\mathcal{S})_{\star\star}$	single	1 $\downarrow$	$\{\mathcal{N}\}$	no	B	yes	no	$y/y(e)$	$\{\mathcal{D}\}$	$\{\mathcal{T}\}$
Backward Cond.	$\mathcal{P}(\mathcal{I})$	$\mathcal{P}(\mathcal{S})_{\star\star}$	single	n $\uparrow$	$\{\mathcal{N}\}$	no	B	yes	no	$y/y(e)$	$\{\mathcal{D}\}$	$\{\mathcal{T}\}$
Pre/Post Cond.	$\mathcal{P}(\mathcal{I})$	$\mathcal{P}(\mathcal{S})_{\star\star}$	single	n $\updownarrow$	$\{\mathcal{N}\}$	no	B	yes	no	$y/y(e)$	$\{\mathcal{D}\}$	$\{\mathcal{T}\}$
Path	$\varnothing$	$\mathcal{P}(\mathcal{S})_{\star\star\star\star}$	single	no	$\{\mathcal{N}\}$	no	B	yes	no	$y/y(e)$	$\{\mathcal{D}\}$	$\{\mathcal{T}\}$
Decomposition	$\mathcal{P}(\mathcal{I})$	$\{\mathcal{S}\}$	n derived	no	$\{\mathcal{N}\}$	no	B	yes	no	y/y	$\{\mathcal{D}\}$	$\{\mathcal{T}\}$
Chopping	$\mathcal{P}(\mathcal{I})$	$\{\mathcal{S}\}$	n+n'	no	$\{\mathcal{N}\}$	no	$B \wedge F$	yes	no	y/n	$\{\mathcal{D}\}$	$\{\mathcal{T}\}$
Relevant	$\mathcal{P}(\mathcal{I})$	single	single	no	single	no	B	no	no	y/y	$\{\mathcal{D}\}$	$\{\mathcal{T}\}$
Hybrid	$\mathcal{P}(\mathcal{I})$	$\mathcal{P}(\mathcal{S})_{\star\star\star}$	1+n	no	$\{\mathcal{N}\}$	no	B	yes	no	$y/y(e)$	$\{\mathcal{D}\}$	$\{\mathcal{T}\}$
Intraprocedural	$\mathcal{P}(\mathcal{I})$	$\{\mathcal{S}\}$	single	no	$\{\mathcal{N}\}$	no	B	no	no	$y/y(e)$	$\{\mathcal{D}\}$	$\{\mathcal{T}\}$
Interprocedural	$\mathcal{P}(\mathcal{I})$	$\{\mathcal{S}\}$	single	no	$\{\mathcal{N}\}$	no	B	yes	no	$y/y(e)$	$\{\mathcal{D}\}$	$\{\mathcal{T}\}$
Simultaneous	$\mathcal{P}(\mathcal{I})$	$\{\mathcal{S}\}$	n	no	$\{\mathcal{N}\}$	no	$B \vee F$	yes	no	$y/y(e)$	$\{\mathcal{D}\}$	$\{\mathcal{T}\}$
Simul. Dyn.	$\mathcal{P}(\mathcal{I})$	n	single	no	$\{\mathcal{N}\}$	no	B	yes	no	$y/y(e)$	$\{\mathcal{D}\}$	$\{\mathcal{T}\}$
Interface	$\mathcal{P}(\mathcal{I})$	$\{\mathcal{S}\}$	n	no	$\{\mathcal{N}\}$	no	B	no	no	$y/y(e)$	$\{\mathcal{D}\}$	$\{\mathcal{T}\}$
Stop-List	$\mathcal{P}(\mathcal{I})$	$\{\mathcal{S}\}$	single	no	$\{\mathcal{N}\}$	no	B	yes	vars	y/n	$\{\mathcal{D}\}$	$\{\mathcal{T}\}$
Barrier	$\mathcal{P}(\mathcal{I})$	$\{\mathcal{S}\}$	single	no	$\{\mathcal{N}\}$	no	B	yes	stats	y/n	$\{\mathcal{D}\}$	$\{\mathcal{T}\}$
Dicing	$\mathcal{P}(\mathcal{I})$	$\{\mathcal{S}\}$	n	no	$\{\mathcal{N}\}$	no	B	yes	no	y/n	$\{\mathcal{D}\}$	$\{\mathcal{T}\}$
Abstract	$\mathcal{P}(\mathcal{I})$	$\mathcal{P}(\mathcal{S})_{\star\star}$	single	$n \updownarrow$	$\{\mathcal{N}\}$	no	B	yes	no	$y/y(e)$	$\{\mathcal{D}\}$	$\{\mathcal{T}\}$
Amorphous	$\mathcal{P}(\mathcal{I})$	$\{\mathcal{S}\}$	single	no	$\{\mathcal{N}\}$	no	B	yes	no	$n/y(e)$	$\{\mathcal{D}\}$	$\{\mathcal{T}\}$
Incremental	$\mathcal{P}(\mathcal{I})$	$\{\mathcal{S}\}$	single	no	$\{\mathcal{N}\}$	no	B	yes	no	y/y	$\mathcal{P}(\mathcal{D})$	$\{\mathcal{T}\}$
Proposition	$\mathcal{P}(\mathcal{I})$	$\{\mathcal{S}\}$	n derived	no	$\{\mathcal{N}\}$	no	B	yes	no	y/y	$\{\mathcal{D}\}$	$\{\mathcal{T}\}+$
New Techniques												
Statement	$\varnothing$	$\{\mathcal{S}\}$	single	no	$\{\mathcal{N}\}$	no	B	no	no	$y/y(e)$	$\{\mathcal{D}\}$	$\{\mathcal{T}\}$
Point	$\varnothing$	$\{\mathcal{S}\}$	single	no	$\{\mathcal{N}\}$	no	B	no	no	$y/y(e)$	$\{\mathcal{D}\}$	$\{\mathcal{T}\}$
Conditioned	$\varnothing$	$\mathcal{P}(\mathcal{S})_{\star\star}$	single	n $\updownarrow$	$\{\mathcal{N}\}$	no	B	no	no	$y/y(e)$	$\{\mathcal{D}\}$	$\{\mathcal{T}\}$
Cond. Chopping	$\mathcal{P}(\mathcal{I})$	$\mathcal{P}(\mathcal{S})_{\star\star}$	pair	n $\updownarrow$	$\{\mathcal{N}\}$	no	$B \wedge F$	yes	no	y/n	$\{\mathcal{D}\}$	$\{\mathcal{T}\}$
Dyn. Chopping	$\mathcal{P}(\mathcal{I})$	$\mathcal{P}(\mathcal{S})_{\star\star}$	pair	n $\updownarrow$	$\{\mathcal{N}\}$	no	$B \wedge F$	yes	no	y/n	$\{\mathcal{D}\}$	$\{\mathcal{T}\}$
Barrier Dicing	$\mathcal{P}(\mathcal{I})$	$\{\mathcal{S}\}$	n	no	$\{\mathcal{N}\}$	no	B	yes	stats	y/n	$\{\mathcal{D}\}$	$\{\mathcal{T}\}$
Filtered	$\mathcal{P}(\mathcal{I})$	$\{\mathcal{S}\}$	single	no	$\{\mathcal{N}\}$	no	B	yes	no	y/n	$\{\mathcal{D}\}$	$\mathcal{P}(\mathcal{T})$
Augmented	$\mathcal{P}(\mathcal{I})$	$\{\mathcal{S}\}$	single	no	$\{\mathcal{N}\}$	no	B	yes	no	y/n	$\{\mathcal{D}\}$	$\{\mathcal{T}\}\pm$
Forw. Amorph.	$\mathcal{P}(\mathcal{I})$	$\{\mathcal{S}\}$	single	no	$\{\mathcal{N}\}$	no	F	yes	no	$n/y(e)$	$\{\mathcal{D}\}$	$\{\mathcal{T}\}$
Forward Point	$\varnothing$	$\{\mathcal{S}\}$	single	no	$\{\mathcal{N}\}$	no	F	no	no	$y/y(e)$	$\{\mathcal{D}\}$	$\{\mathcal{T}\}$

$\star$ all agree on input prefix $\star\star$ all agree on conditions $\star\star\star$ all agree on breakpoints $\star\star\star\star$ all agree on path

Here, statement (1) cannot influence the value of y at statement (3); however, statement (1) would be included by almost all slicing techniques computed with respect to $\langle 3, y\rangle$, because statement (1) influences statement (2), which in turn influences statement (3).

Table I extends Harman et al.'s [1996] classification with new dimensions in order to be able to classify new techniques. In the table, we have omitted those slicing criteria which have been identified as particular cases of another technique in Section 2 and, thus, do not impose additional restrictions or dimensions in the table. For instance, we omit end slicing in the table because it is a particular case of simultaneous slicing; we also omit call-mark slicing and dependence-cache slicing because they are versions of dynamic slicing where the dynamic data is handled differently. In addition, we only include the most general form of slicing for each technique. For instance, in the case of quasi-static slicing, the original definition by Venkatesh [1991] considered slicing only "at the end of the program," here, in contrast, we consider a generalization (see Section 2.11) in which any point of the program can be selected for slicing.

In the table, following Harman et al.'s terminology, the sets $\mathcal{I}$, $\mathcal{S}$, $\mathcal{N}$, $\mathcal{D}$, and $\mathcal{T}$ refer, respectively, to the set of all program variables, possible initial states, possible iterations, program statements, and types of dependencies.

—Column Variables (vars) specifies the number of variables participating in a slicing criterion $\mathcal{C}$, where $\mathcal{P}(\mathcal{I})$ indicates that a subset of $\mathcal{I}$ participates in $\mathcal{C}$.
—Column Initial States shows the number of initial states considered by the corresponding slicing criterion.
—Column Slice Points describes the number of program points included in $\mathcal{C}$, and hence, the number of slices actually produced (and combined) to extract the final slice. Here, "n derived" means that n different points are implicitly specified (see Section 2.25); "1+n" means that 1 program point is specified together with n breakpoints, and "n+n'" means that n points are specified for the source, and n' points are specified for the sink,
—Column Statements Conditions (sta. Con.) specifies if the statements in the slice must fulfill any condition. Here, we use a number to describe how many conditions can be specified, and we use an arrow to describe the direction of the condition: $\downarrow$ indicates that the conditions affects the statements after the condition, $\uparrow$ indicates that the conditions affects the statements before the condition, and $\updownarrow$ indicates that the conditions can be specified for both directions.
—Column Iteration Counts (Iter. Cou.) is related to the fact that the slicing point can be executed several times (e.g., if it is inside a loop or a procedure), thus producing several values for its variables. Hence, iteration counts indicates, for the specified slice points, which iterations are of interest.
—Column Path Aware (Pa. Aw.) indicates if the slice produced is path aware, as in Korel and Laski's style of slicing [1988] (see Section 2.3), or it is path unaware.
—Column Direction (Dir.) states the direction of the traversal to produce the slice which can be backwards (B), forwards (F), or a combination of both. Here, despite that some techniques accept both directions, we assign to every technique the direction specified in its original definition (see Section 2).
—Column Precision (Pr.) is marked if the slice produced is precise.
—Column Limits (Lim.) specifies if the computation of the slice is limited or not. Here, "vars" specifies that a list of variables is used as the limit; and "stats" specifies that a list of statements is used as the limit.
—Column Syntax/Semantics Preserving (Sy/Se. Pres.) contains boolean values that indicate, respectively, whether the slice produced is a projection of the program syntax or it preserves a projection of the original semantics. Here, "(e)" means that the slice produced is executable.
—Column Dependences Considered (Dep. Con.) specifies how many types of dependences are considered to compute the slice.
—Column Statements Considered (Sta. Cons.) specifies what statements could be part of the slice. Here, "+" indicates that a subset of statements are included in the slice even if they do not contribute to the slicing criterion; whereas "−" indicates that a subset of statements are not included in the slice even if they do contribute to the slicing criterion.

For precision in the slicing criteria comparison, we have included in the table two versions of dynamic slicing. The first one is the original definition by Korel and Laski [1988] which is path-aware; the second one is the path-unaware definition introduced by Agrawal and Horgan [1990]. We put attention on Agrawal and Horgan's definition because it is more widely used in the program slicing community, and thus, it is

Table II. Binkley et al.'s Slicing Techniques Produced by Permutation of Slicing Dimensions

Slicing Technique	Initial States	Iteration Count	Path Aware
Static Slicing (SS)	$\{S\}$	no	no
Dynamic Slicing (DS)	single	no	no
Static Iteration Count Slicing (SIS)	$\{S\}$	yes	no
Dynamic Iteration Count Slicing (DIS)	single	yes	no
Static Path-Aware Slicing (SPS)	$\{S\}$	no	yes
Dynamic Path-Aware Slicing (DPS)	single	no	yes
Static Iteration Count Path-Aware Slicing (SIPS)	$\{S\}$	yes	yes
Dynamic Iteration Count Path-Aware Slicing (DIPS)	single	yes	yes

comparable to most slicing techniques. In the case of simultaneous slicing, for concreteness, we have assumed that the criteria combined are static slicing criteria.

Each dimension in the table introduces a way to do slicing which, in general, can be adapted to all the techniques. Consider for instance the Direction dimension. Each technique has been assigned a value for this dimension (i.e., the value that the author used when the technique was defined); however, other values can be used giving rise to different forms of slicing. For instance, we can think of backward static slicing and forward static slicing; backward barrier slicing and forward barrier slicing, etc. Similarly, the technique amorphous slicing introduced a new dimension (i.e., a new way to do slicing), and thus, it could be applied to all the slicing techniques (e.g., amorphous conditioned slicing vs. non-amorphous conditioned slicing). The way in which the slice is computed can be different too. For instance, we could think of intraprocedural barrier slicing and interprocedural barrier slicing. Consequently, each dimension in the table is a potential source of slicing techniques, and new forms of slicing appear when either a new value for a dimension is defined or when a new dimension is discovered. In contrast, a new slicing technique appears in the literature when a new combination of values for the dimensions is found useful. Note, we are assuming here that the application of a technique in a different language, paradigm, or context in general is not a new slicing technique but an adaptation of an existing slicing technique. Under this assumption, Table I is a mighty tool that can be used to determine the degree of novelty of new slicing techniques.

The slicing criteria of the table summarize the collection of slicing-based techniques produced during the last 30 years. As explained in Section 2, all these criteria have been published, implemented, and applied to different software engineering fields. However, of course, by combining different values for the dimensions, we could produce several new slicing techniques. Unfortunately, in general, the techniques produced with this method would be useless in practice.

Binkley et al. [2006a, 2006b] studied the techniques produced by permuting the values of three dimensions in the table, namely 'Initial States', 'Iteration Counts', and 'Path-Aware'. The result is the eight techniques shown in Table II.

The comparison of slicing techniques presented in Table I is a powerful tool for the study of current slicing techniques and the prediction of new ones. The last ten rows in Table I correspond to new slicing techniques predicted by analyzing the information of the table. These new techniques have been predicted by trying to introduce a facility of a technique into another technique to increase its power.

The first new technique is statement slicing. This technique answers the question, which statements can possibly influence reachability of statement s? Surprisingly, this question cannot be answered by previous slicing techniques. A statement slice is composed of all the possible path slices of a program with respect to a given statement.

```
(1)   read(text);
(6)   c = getChar(text);
(7)   while (c != '\eof')
(8)         if (c == '\n')
(9)         then lines = lines + 1;
(15)        c = getChar(text);
```

Fig. 28. Statement slice of the program in Figure 1(a) with respect to statement (9).

That is, statement slicing is a generalization of path slicing where all the possible paths are considered.[4]

For instance, consider the program in Figure 1(a). We want to know what statements can possibly influence reachability of statement (18) when the input is finite. The answer is none (the statement slice would be empty), because this statement will be executed always. This is rather different from the slice produced by other slicing techniques which focus on the value of a variable. Here, as in path slicing, the variables of interest are those which belong to conditions that can influence reachability of the statement of interest. In this example, computing a path slice is as easy as looking at the PDG where we see that the control of the program will always arrive to statement (18). In general, to compute the statement slice, it is necessary to check which statements can influence the conditions that determine whether the point of interest is going to be executed or not. For instance, consider now the same program and the statement (9). In this case, the statement slice produced is shown in Figure 28.

Statements (7) and (8) are needed to execute (9) (due to a control dependency); statements (6) and (15) are needed to execute (7); and (1) to execute (6) (due to a data dependency).

Another new technique is point slicing. Point slicing tries to answer the question, What statements could have been executed before statement s? Again, this question could not be answered by previous techniques.

As with statement slicing, point slicing considers a single statement in the slicing criterion. Point slicing selects a statement of the program and computes backwards (respectively, forwards) all the statements that could have been executed before (respectively, after) it. Apart from its clear application in program comprehension, this technique can be useful in debugging. For instance, during print debugging [Agrawal 1991], the programmer places print statements in some strategic points of the program and then executes it in order to see if some of them have been reached and in which order they have been executed. Usually, a print statement is placed in a part of the code in order to check if it is executed or not. A point slice with respect to this statement can be useful in order to know which possible paths of the program could reach the execution of this statement. Note that point slicing does not follow control or data dependencies, but control flows; thus, it is not subsumed by any of the other slicing techniques in the table. As an example, the statement `print ('Executed');` does not influence any other statement, and hence, it will not belong to any slice taken with respect to a subset of the variables of the program (it will be removed in all the slicing techniques!). The implementation of this technique is straightforward, because a point slice can be constructed by traversing the control flow graph from the point of interest.

The conditioned version of point slicing restricts the possible paths by limiting the initial states of the execution. By using the pre/postconditions of pre/postconditioned slicing, the programmer would be able to include conditions inside the source code. This facility would significantly increase the power of the technique, because it would allow

[4]We could also restrict the number of paths by using a condition. This would lead us to conditioned statement slicing.

the programmer to know which paths of the program can be executed before a statement in a particular context (e.g., can statement s_1 be executed before statement s_2 if this flag is activated?). This would significantly reduce the work of a print debugging user by automatizing much of the work. Of course, the implementation of pre/postconditioned point slicing is not trivial as point slicing is.

Conditioned chopping is a very general form of slicing which subsumes both conditioned slicing and chopping, and hence, dynamic chopping (neither are defined yet). Dynamic chopping can be useful, for instance, in debugging. When tracing a computation, a programmer usually proceeds (big) step by step until a wrong subcomputation is found (e.g., a procedure call that returned a wrong result). Dynamic chopping comes in handy at this point because it can compute the statements that influenced the wrong result from the procedure call. This dynamic chop is, in general, much smaller than the static chop, because it only considers a particular execution.

It should be clear that the source and the sink of chopping are not conditions but points. Therefore, conditioned chopping computes all the statements in the program that, being affected by source, affect sink when some pre/postconditions are satisfied.

Barrier dicing is a new form of dicing in which barriers are used to eliminate from the slice those parts that the user knows are correct. Once an incorrectly valued variable has been found and a slice produced for it, the objective of dicing is to reduce this slice by eliminating the statements appearing in a static slice of variables that appear to be correctly computed. However, in the slice, there still remain many statements which the programmer knows are correct (e.g., reused procedures and functions, legacy code, etc.). The barrier facilities of barrier slicing are useful at this point, because they can be set at strategic points (e.g, procedure calls) to avoid inclusion in the slice of correct code; thus increasing the power of dicing.

Filtered slicing generalizes static slicing. While the other techniques collect all statements that influence the slicing criterion, filtered slicing only collects the subset of statements that influence the slicing criterion and that are of a given type. Clearly, the idea behind filtered slicing is similar to the one of incremental slicing and thin slicing. These techniques only consider a type of dependencies, and filtered slicing only considers a type of statement. A filtered slicing criterion is a triple $\langle s, v, c\rangle$, where s and v keep the standard meaning, and c specifies a condition that statements must fulfill in order to be part of the slice. This freedom allows the user to produce smaller slices that only focus on a particular comprehension task. For instance, the filtered slice of the program in Example 1(a) with respect to the slicing criterion $\langle 16, lines, s$ defines variable $lines\rangle$ contains statements (3) and (9). This slice shows what assignments to $lines$ influence the slicing criterion. The filtered slice with respect to $\langle 16, lines, s$ contains $c\rangle$ contains statements (6), (7), (8), and (15). This slice shows only the contribution of c to the computation of the slicing criterion. The filtered slice with respect to $\langle 16, lines, s$ is an if or a $while$ statement$\rangle$ contains statements (7) and (8). This slice shows all the conditions that can be traversed before reaching the slicing criterion.

Augmented slicing generalizes proposition-based slicing by allowing us to exclude statements from the slice. Moreover, in contrast to proposition-based slicing where the statements added to the slice are predefined, in augmented slicing, the statements added are specified by the user. An augmented slicing criterion is a tuple $\langle s, v, c, f\rangle$, where s and v keep the standard meaning of static slicing, c specifies a condition that statements must fulfill in order to be part of the slice, and f specifies a condition that statements must fulfill in order to be excluded from the slice.

Finally, the last two rows of the table—forward amorphous and forward point slicing—correspond to the forward versions of these techniques which should be investigated as a program comprehension tool. On the one hand, forward amorphous collects those statements affected by the slicing criterion but without the syntax preservation

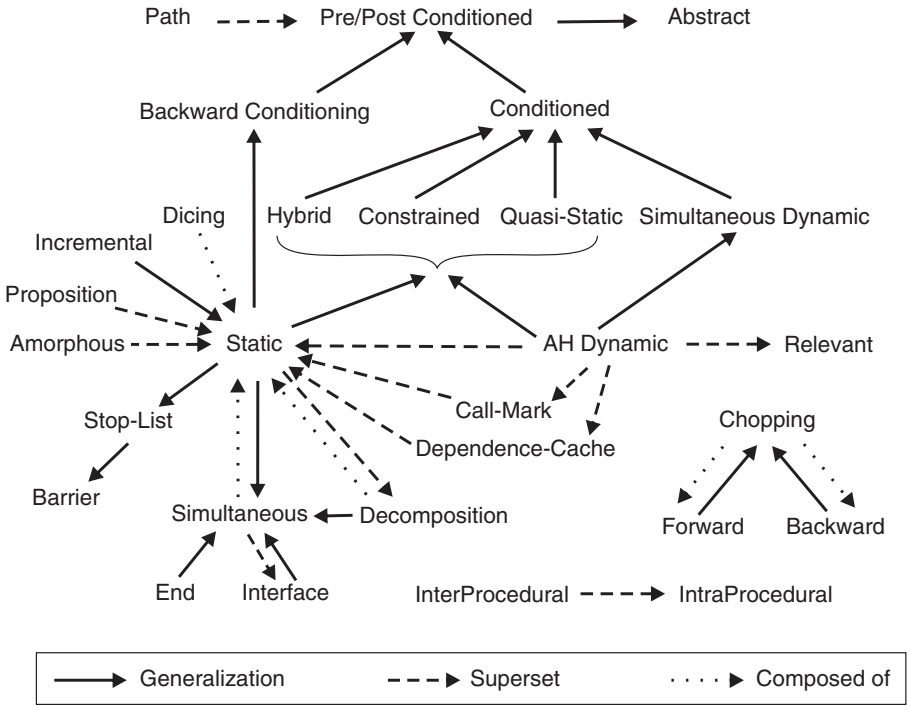


Fig. 29. Relationships between program slicing techniques.

restriction. This allows us to present to the programmer more compact and understandable slices. On the other hand, forward point slicing allows us to answer the question, What statements could be executed after statement s? Here, again, the use of pre/postconditions would increase the power of this technique.

3.2. Interrelations Between Slicing Techniques

The information in Table I can also be used to identify relations between slicing techniques. We have identified some relations and represented them in the graph of Figure 29, considering that each pair of related slicing techniques relate comparable slicing criteria. There are three kinds of arrows in the graph.

Generalization ($S_1 \longrightarrow S_2$). A slicing technique S_2 generalizes another technique S_1 if and only if all the slicing criteria that can be specified with S_1 can also be specified with S_2. This relation answers the question, is slicing technique A a particular case of slicing technique B?

Superset ($S_1 \dashrightarrow S_2$). The slice produced by a slicing technique S_2 is a superset of the slice produced by another technique S_1 if and only if all the statements in S_1 also belong to S_2. This relation answers the question, *is the slice*[5] *produced by slicing technique A included in the slice produced by slicing technique B?*

Composed of ($S_1 \cdots\triangleright S_2$). A slicing technique S_1 is composed of the technique S_2 if and only if the slicing criterion of S_1 contains the slicing criterion of S_2. For instance, chopping is composed of two slicing techniques (forward slicing and backward slicing),

[5]Recall that we are referring to minimal slices.

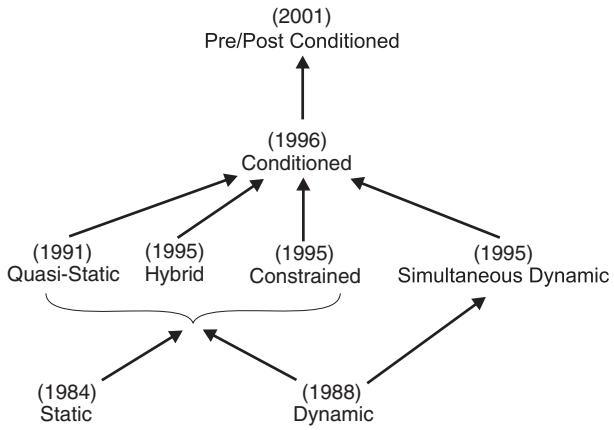


Fig. 30. Generalization relationships between program slicing techniques which develop the *initial states* dimension.

dicing is composed of n slicing techniques (usually one backward slice of an incorrect value, and $n-1$ backward slices of a correct value), and simultaneous static slicing is composed of n slicing techniques. In the graph, for clarity, we use only static slicing as the component of dicing and simultaneous static slicing. This relation answers the question, is slicing technique A used by slicing technique B?

The graph in Figure 29 mixes together all the techniques. Each node in the graph represents a technique of Table I. Therefore, each node has a different parameterization for the dimensions in Table I, and the reader should not confuse techniques with particular values of dimensions. For instance, the node "Forward" represents the technique "Forward Slicing" described in Section 2.5 rather than the value "Forward" of the "Direction" dimension. Therefore, this graph must be complemented with Table I in order to know the exact meaning of each arrow. The graph can say that technique A generalized technique B, but it cannot explain why. For instance, incremental slicing generalizes static slicing by allowing us to only consider a subset of types of dependences. This information comes from Table I. Note that the graph naturally places static slicing (Weiser's technique) as the core of the relations. This also allows us to see how this technique has evolved in many directions.

If we concentrate on a subset of the dimensions, we can extract useful information. For instance, if we only focus on the Initial states dimension, we get the graph in Figure 30. This graph only contains those techniques which have concentrated on restricting the initial states considered. In the figure, we have included the year when each technique was published, and thus, we see that this dimension was exploited during the 90's.

We can also focus on a set of dimensions and get interesting conclusions, Binkley et al. [2006a, 2006b] have done with the eight techniques of Table II. They used the superset relation to relate different permutations of the values of three dimensions, producing the graph in Figure 31. Since each pair of values taken in the same dimension imply a less or more restricted slice, their combination produces a lattice.

The classification points out abstract slicing as one of the most general slicing techniques, thanks to its use of predicates and conditions. While chopping generalizes forward and backward slicing, an abstract chop [Hong et al. 2005] generalizes forward and backward abstract slicing. Hence, abstract chopping subsumes static, dynamic, backward, and forward slicing.

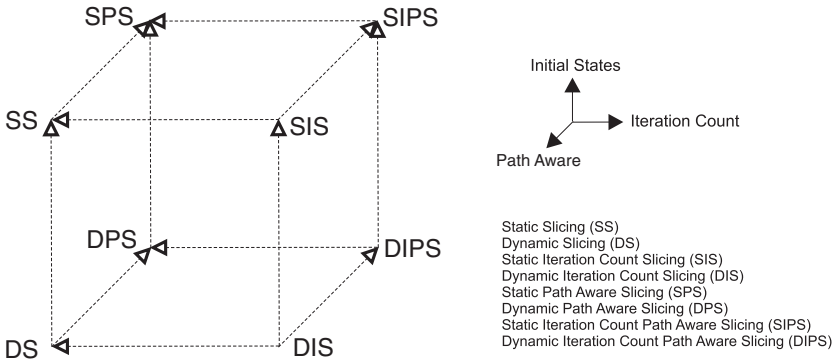


Fig. 31. Superset relationships between the program slicing techniques of Table II.

4. CONCLUSIONS

During the last thirty years, program slicing has been applied to solve a wide variety of problems. Each application required a different extension, generalization, or combination of previous slicing techniques. In this work, we have described and compared these techniques in order to classify them according to their properties and in order to establish a hierarchy of slicing techniques. In particular, we have extended a comparative table of slicing techniques by Harman et al. [1996] with new dimensions and techniques. The classification of techniques presented not only shows the differences between them; it also allows us to predict not yet used future techniques in the table that will fit a combination of parameters. By combining strong points of different slicing techniques, we can predict a more powerful technique that can solve problems not addressed before.

With the information provided in this classification, we have studied and identified three kinds of relations between the techniques: composition, generalization, and superset relations. This study allows us to reason about the interrelations between slicing techniques; to observe how (in which order and when) the techniques have been developed, thus reasoning about the evolution of program slicing; and to answer questions like, is one technique more general than another technique? Is the slice produced by one technique included in the slice of another technique? Is one technique composed of other techniques?

ACKNOWLEDGMENTS

The author wishes to thank Germán Vidal for many useful comments on the contents of this article, and for kindly reviewing its preliminary version. He also wishes to thank the anonymous reviewers for their thorough readings of the paper and their thoughtful suggestions.

REFERENCES

AGRAWAL, H. 1991. Towards automatic debugging of computer programs. Tech. rep. Ph.D. dissertation, Purdue University, Indiana.

AGRAWAL, H. AND HORGAN, J. R. 1990. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 246–256.

AGRAWAL, H., HORGAN, J. R., KRAUSER, E. W., AND LONDON, S. 1993. Incremental regression testing. In *Proceedings of the Conference on Software Maintenance*. IEEE Computer Society, 348–357.

BECK, J. 1993. Interface slicing: A static program analysis tool for software engineering. Ph.D. dissertation, Morgantown, WV.

BECK, J. AND EICHMANN, D. 1993. Program and interface slicing for reverse engineering. In *Proceedings of the 15th International Conference on Software Engineering*. IEEE Computer Society Press, 509–519.

Bergeretti, J. and Carré, B. 1985. Information-flow and data-flow analysis of While-programs. *ACM Trans. Program. Lang. Syst. 7,* 1, 37–61.

Beszédes, Á., Faragó, C., Szabó, Z. M., Csirik, J., and Gyimóthy, T. 2002. Union slices for program maintenance. In *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society, 12–21.

Binkley, D., Danicic, S., Gyimóthy, T., Harman, M., Kiss, Á., and Korel, B. 2006a. A formalization of the relationship between forms of program slicing. *Sci. Comput. Program*. *62,* 3, 228–252.

Binkley, D., Danicic, S., Gyimóthy, T., Harman, M., Kiss, A., and Korel, B. 2006b. Theoretical foundations of dynamic program slicing. *Theoretical Computer Science 360,* 1, 23–41.

Binkley, D., Danicic, S., Gyimothy, T., Harman, M., Kiss, A., and Ouarbya, L. 2004. Formalizing executable dynamic and forward slicing. In *Proceedings of the 4th IEEE International Workshop on Source Code Analysis and Manipulation*. IEEE Computer Society, 43–52.

Binkley, D. and Gallagher, K. B. 1996. Program slicing. *Adv. Comput. 43*, 1–50.

Binkley, D. and Harman, M. 2004. A survey of empirical results on program slicing. *Adv. Comput. 62*, 105–178.

Canfora, G., Cimitile, A., and De Lucia, A. 1998. Conditioned program slicing. *Inform. Softw. Technol. 40,* 11–12, 595–608.

Canfora, G., Cimitile, A., De Lucia, A., and Lucca, G. A. D. 1994. Software salvaging based on conditions. In *Proceedings of the International Conference on Software Maintenance*. 424–433.

Chen, T. Y. and Cheung, Y. Y. 1993. Dynamic program dicing. In *Proceedings of the International Conference on Software Maintenance*. 378–385.

Cheney, J. 2007. Program slicing and data provenance. *IEEE Data Engineer. Bullet. 30,* 4, 22–28.

Cheng, J. 1993. Slicing concurrent programs—A graph-theoretical approach. In *Proceedings of the International Workshop on Automated and Algorithmic Debugging*. 223–240.

Danicic, S., Fox, C., Harman, M., Hierons, R., Howroyd, J., and Laurence, M. 2005. Static program slicing algorithms are minimal for free liberal program schemas. *Comput. J. 48,* 6, 737–748.

Danicic, S. and Harman, M. 1996. Program slicing using functional networks. In *Proceedings of the 2nd U.K. Workshop on Program Comprehension*. 54–65.

De Lucia, A. 2001. Program slicing: Methods and applications. In *Proceedings of the 1st IEEE International Workshop on Source Code Analysis and Manipulation*. IEEE Computer Society Press, Los Alamitos, California, 142–149.

De Lucia, A., Fasolino, A. R., and Munro, M. 1996. Understanding function behaviors through program slicing. In *Proceedings of the 4th Workshop on Program Comprehension*, A. Cimitile and H. A. Müller, Eds. IEEE Computer Society Press, 9–18.

De Lucia, A., Harman, M., Hierons, R. M., and Krinke, J. 2003. Unions of slices are not slices. In *Proceedings of the 7th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, 363–367.

Dwyer, M. and Hatcliff, J. 1999. Slicing software for model construction. In *Proceedings of the Symposium on Partial Evaluation and Semantic-Based Program Manipulation*. 105–118.

Ferrante, J., Ottenstein, K., and Warren, J. 1987. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst. 9,* 3, 319–349.

Field, J., Ramalingam, G., and Tip, F. 1995. Parametric program slicing. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, 379–392.

Fox, C., Harman, M., Hierons, R., and Danicic, S. 2001. Backward conditioning: A new program specialization technique and its application to program comprehension. In *Proceedings of the 9th IEEE International Workshop on Program Comprehension* . 89–97.

Gallagher, K., Binkley, D., and Harman, M. 2006. Stop-list slicing. In *Proceedings of the 6th IEEE International Workshop on Source Code Analysis and Manipulation*. 11–20.

Gallagher, K. and Lyle, J. 1991. Using program slicing in software maintenance. *IEEE Trans. Softw. Engin. 17,* 8, 751–761.

Gallagher, K. B. 2004. Some notes on interprocedural program slicing. In *Proceedings of the IEEE International Workshop on Source Code Analysis and Manipulation*. IEEE Computer Society, 36–42.

Gaucher, F. 2003. Slicing Lustre programs. Tech. rep., VERIMAG, Grenoble.

Giacobazzi, R. and Mastroeni, I. 2003. Non-standard semantics for program slicing. *Higher Order Symbol. Comput. 16,* 4, 297–339.

Giffhorn, D. and Hammer, C. 2007. An evaluation of slicing algorithms for concurrent programs. In *Proceedings of the 7th IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE Computer Society, 17–26.

GREIBACH, S. 1985. *Theory of Program Structures: Schemes, Semantics, Verification*. Springer-Verlag New York, Inc., Secaucus, NJ.

GUPTA, R. AND SOFFA, M. 1995. Hybrid slicing: An approach for refining static slices using dynamic information. In *Proceedings of the 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM Press, 29–40.

GYIMÓTHY, T., BESZÉDES, Á., AND FORGÁCS, I. 1999. An efficient relevant slicing method for debugging. In *Proceedings of the 7th European Software Engineering Conference*. Springer-Verlag, 303–321.

HALL, R. 1995. Automatic extraction of executable program subsets by simultaneous dynamic program slicing. *Auto. Softw. Engin. 2,* 1, 33–53.

HARMAN, M. AND DANICIC, S. 1997. Amorphous program slicing. In *Proceedings of the 5th International Workshop on Program Comprehension*. IEEE Computer Society Press, 70–79.

HARMAN, M., DANICIC, S., SIVAGURUNATHAN, Y., AND SIMPSON, D. 1996. The next 700 slicing criteria. In *Proceedings of the 2nd U.K. Workshop on Program Comprehension*.

HARMAN, M. AND GALLAGHER, K. 1998. Program slicing. *Inform. Softw. Technol. 40,* 11–12, 577–582.

HARMAN, M. AND HIERONS, R. 2001. An overview of program slicing. *Softw. Focus 2,* 3, 85–92.

HARMAN, M., HIERONS, R. M., FOX, C., DANICIC, S., AND HOWROYD, J. 2001. Pre/postconditioned slicing. In *Proceedings of the International Conference on Software Maintenance*. 138–147.

HONG, H., LEE, I., AND SOKOLSKY, O. 2005. Abstract slicing: A new approach to program slicing based on abstract interpretation and model checking. In *Proceedings of the 5th IEEE International Workshop on Source Code Analysis and Manipulation*. IEEE Computer Society, 25–34.

HORWITZ, S., REPS, T., AND BINKLEY, D. 1988. Interprocedural slicing using dependence graphs. In *Proceedings of the Conference on Programming Language Design and Implementation*, ACM SIGPLAN, *23*, 7, 35–46.

JACKSON, D. AND ROLLINS, E. 1994. Chopping: A generalization of slicing. Tech. rep. CS-94-169, Carnegie Mellon University, School of Computer Science.

JHALA, R. AND MAJUMDAR, R. 2005. Path slicing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM Press, New York, NY, 38–47.

JONES, N. 1996. An introduction to partial evaluation. *ACM Comput. Surv. 28,* 3, 480–503.

KOREL, B. AND LASKI, J. 1988. Dynamic program slicing. *Inform. Process. Letters 29,* 3, 155–163.

KRINKE, J. 1998. Static slicing of threaded programs. *SIGPLAN Notices 33,* 7, 35–42.

KRINKE, J. 2003a. Barrier slicing and chopping. In *Proceedings of the 3rd IEEE International Workshop on Source Code Analysis and Manipulation*. IEEE Computer Society, 81–87.

KRINKE, J. 2003b. Context-sensitive slicing of concurrent programs. In *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering*. 178–187.

KRINKE, J. 2004. Slicing, chopping, and path conditions with barriers. *Softw. Qual. J. 12,* 4, 339–360.

KUCK, D., KUHN, R., PADUA, D., LEASURE, B., AND WOLFE, M. 1981. Dependence graphs and compiler optimization. In *Proceedings of the 8th Symposium on the Principles of Programming Languages, SIGPLAN Notices*. 207–218.

LAKHOTIA, A. 1993. Rule-based approach to computing module cohesion. In *Proceedings of the 15th International Conference on Software Engineering*. IEEE Computer Society Press, Los Alamitos, CA, 35–44.

LARSEN, L. AND HARROLD, M. 1996. Slicing object-oriented software. In *Proceedings of the 18th International Conference on Software Engineering*. IEEE Computer Society, 495–505.

MÜLLER-OLM, M. AND SEIDL, H. 2001. On optimal slicing of parallel programs. In *Proc. of the 33rd ACM Symposium on Theory of Computing*. ACM, New York, NY, 647–656.

NANDA, M. AND RAMESH, S. 2000. Slicing concurrent programs. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, New York, NY, 180–190.

LYLE, J. AND WEISER, M. 1987. Automatic program bug location by program slicing. In *Proceedings of the 2nd International Conference on Computers and Applications*. 877–882.

NING, J., ENGBERTS, A., AND KOZACZYNSKI, W. 1994. Automated support for legacy code understanding. *Commun. ACM 37,* 5, 50–57.

NISHIMATSU, A., JIHIRA, M., KUSUMOTO, S., AND INOUE, K. 1999. Call-mark slicing: An efficient and economical way of reducing slices. In *Proceedings of the 21st International Conference on Software Engineering*. ACM Press, 422–431.

ORSO, A., SINHA, S., AND HARROLD, M. 2001a. Effects of pointers on data dependences. In *Proceedings of the 9th International Workshop on Program Comprehension (IWPC'01)*. 39–49.

ORSO, A., SINHA, S., AND HARROLD, M. 2001b. Incremental slicing based on data-dependence types. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*. 158–167.

OTTENSTEIN, K. AND OTTENSTEIN, L. 1984. The program dependence graph in a software development environment. In *Proceedings of the 1st ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. ACM Press, New York, NY, 177–184.

REPS, T. AND BRICKER, T. 1989. Illustrating interference in interfering versions of programs. *SIGSOFT Softw. Engineer. Notes 14,* 7, 46–55.

REPS, T. AND ROSAY, G. 1995. Precise interprocedural chopping. *SIGSOFT Softw. Engineer. Notes 20,* 4, 41–52.

SIVAGURUNATHAN, Y., HARMAN, M., AND DANICIC, S. 1997. Slicing, I/O, and the implicit state. In *Proceedings of the International Workshop on Automated and Algorithmic Debugging*. 59–68.

SPARUD, J. AND RUNCIMAN, C. 1997. Tracing lazy functional computations using redex trails. In *Proceedings of the 9th International Symposium on Programming Languages, Implementations, Logics and Programs*. Springer, Berlin, Lecture Notes in Computer Science, vol. 1292, 291–308.

SRIDHARAN, M., FINK, S., AND BODIK, R. 2007. Thin slicing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, NY, 112–122.

TAKADA, T., OHATA, F., AND INOUE, K. 2002. Dependence-cache slicing: A program slicing method using lightweight dynamic information. In *Proceedings of the 10th International Workshop on Program Comprehension*. IEEE Computer Society, 169–177.

TAN, H. AND LING, T. 1998. Correct program slicing of database operations. *IEEE Softw. 15,* 2, 105–112.

TIP, F. 1995. A survey of program slicing techniques. *J. Program. Lang. 3*, 121–189.

VENKATESH, G. A. 1991. The semantic approach to program slicing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, New York, NY, 107–119.

WARD, M. 2002. Program slicing via fermat transformations. In *Proceedings of the 26th International Computer Software and Applications Conference, Prolonging Software Life: Development and Redevelopment*. IEEE Computer Society, 357–362.

WARD, M. 2003. Slicing the scam mug: A case study in semantic slicing. In *Proceedings of the 3rd IEEE International Workshop on Source Code Analysis and Manipulation*. IEEE Computer Society, 88–97.

WARD, M. AND ZEDAN, H. 2007. Slicing as a program transformation. *ACM Trans. Program. Lang. Syst. 29,* 2.

WEISER, M. 1979. Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method. Ph.D. dissertation, University of Michigan.

WEISER, M. 1981. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*. 439–449.

WEISER, M. 1984. Program slicing. *IEEE Trans. Softw. Engineer. 10,* 4, 352–357.

WILLMOR, D., EMBURY, S., AND SHAO, J. 2004. Program slicing in the presence of a database state. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*. IEEE Computer Society, 448–452.

XU, B., QIAN, J., ZHANG, X., WU, Z., AND CHEN, L. 2005. A brief survey of program slicing. *SIGSOFT Softw. Engineer. Notes 30,* 2, 1–36.

ZHAO, J. 2002. Slicing aspect-oriented software. In *Proceedings of the 10th International Workshop on Program Comprehension*. IEEE Computer Society, 251.

Received October 2006; revised December 2007, June 2008; accepted October 2010