An Analysis of the Current Program Slicing and Algorithmic Debugging Based Techniques

JOSEP SILVA
Technical University of Valencia

Advisor: Germán Vidal

December, 2008

2 ·

PART I
Program Slicing Techniques

1. INTRODUCTION

Program slicing was originally introduced in 1984 by Mark Weiser. Since then, many researchers have extended it in many directions and for all programming paradigms. The huge amount 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, [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] focuses on their implementation comparing empirical results; and [Harman et al. 1996] tries 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?*

There is a well-developed theory and formalization of program slicing which allows to formally reason about the semantic implications of different slicing techniques (see e.g., [Venkatesh 1991; Giacobazzi and Mastroeni 2003; Binkley et al. 2004; Danicic et al. 2005; Binkley et al. 2006a; Binkley et al. 2006b; Ward and Zedan 2007]). This formal framework can be used, e.g., 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.

The article has been structured in two parts: In the first part (Section 2) we describe all the techniques in a way similar to [De Lucia 2001]. We propose a running example program and we 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 for each technique at least one slice generated from a slicing criterion which is comparable to the slicing criterion used in the other techniques. The main objective in this part 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 to produce the slices of each technique. We will rather discuss for each technique what information is needed to produce a slice, what peculiarities has this slice with respect to other techniques, and what are its main applications. Of course, we will refer the interested reader to the sources where the techniques were defined and where deeper details about its implementation can be found.

This part may be useful for a researcher (not necessarily a program slicing expert) which is looking for a program slicing technique and she wants to know which of the wide variety of slicing techniques better fits her necessities. To ease her 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] a decade ago and we extend it with new slicing techniques and dimensions. The 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 PhD thesis of Mark Weiser in 1979 [Weiser 1979] that were presented in the *International Conference on Software Engineering* in 1981 [Weiser 1981] and finally published in *Transactions on Software Engineering* in 1984 [Weiser 1984].

Program slicing is a technique to decompose programs by analyzing their data and control flow. Roughly speaking, a *program slice* consists of those program statements 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 been sometimes relaxed. Given a program p, slices are produced w.r.t 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 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 slice, consider again 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

```
*(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 slice

$\langle 16, lines \rangle$ is shown in Figure 1 (b).

Together with the running example, we will use a "running slicing criterion" that we will use 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 to 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.

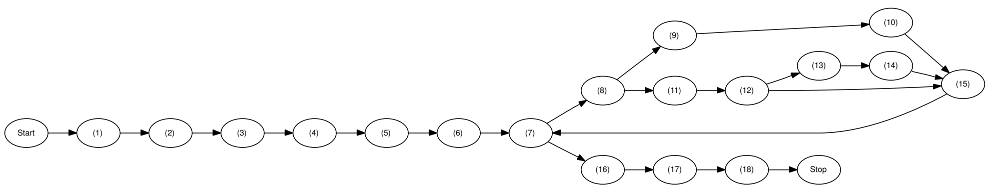


Fig. 2. Control Flow Graph of Figure 1 (a)

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).

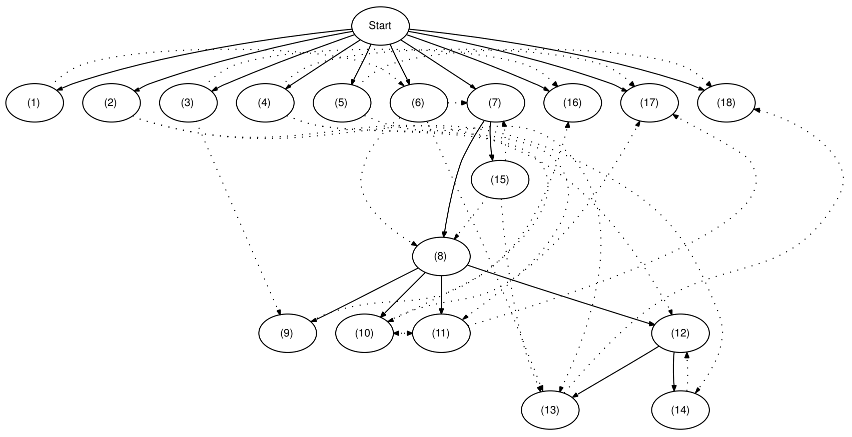


Fig. 3. Program Dependence Graph of Figure 1 (a)

In order to extract a slice from a program, the dependences between its statements must be computed first. The *Control Flow Graph* (CFG) is a data structure which makes the control dependences 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 dependences and data dependences are also necessary. For this reason, the CFG must be annotated with data flow information by marking at each node the set of variables defined and referenced. 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 dependences for each operation in a program. In essence, a PDG is an oriented graph where the nodes represent statements in the source code and edges represent control and data flow dependences 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 dependences 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?

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

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.

2.3 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, i.e., 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 it 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 the set $\{a_1, \ldots, a_n\}$ are 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 dependences produced by the provided input data. However, it comes with a cost: the computation of such dependences usually implies the computation of an expensive (measured in time and space) and complex data structure (e.g., a trace [Sparud and Runciman 1997]).

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

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

[2]This notation has been used in two different ways in the literature. We use the original definition defined in [Korel and Laski 1988] and later user 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*"; it was defined in [Agrawal and Horgan 1990] and later user for instance in [Binkley and Gallagher 1996]".

In their definition of dynamic slicing, Korel and Laski introduced one important novelty: their slices must follow identical paths than its 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. Formally,

Definition 2.1 *Korel and Laski's dynamic slice [Korel and Laski 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, produces a trajectory T' for which there exists an execution position i' such that

(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 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 (see, e.g., [Binkley et al. 2006a]) and that determines whether dynamic slicing is subsumed or not 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 paper—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, see [Agrawal and Horgan 1990]). We refer the reader to Section 3 for a description of slicing dimensions and their use to compare 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 (the so called backward slicing), we are interested in all those statements that could influence the slicing criterion. In contrast, when we traverse it forwards (the so called forward slicing), we are interested in all those statements that could be influenced by the slicing criterion.

The original method by Weiser—described above—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).

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 slice [Reps and Bricker 1989]. However, despite backward slicing is 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 $\langle 3, \{lines\}\rangle$ shown in Figure 5.

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

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

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, they 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 [Reps and Rosay 1995], chopping is particularly useful to detect 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?

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

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

Main applications: Program analysis and debugging.

2.7 Relevant Slicing *(Agrawal, Horgan, Krauser and London, 1993)*

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

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

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

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 in the computation of these large structures needed for dynamic slicing, a new technique called hybrid slicing [Gupta and Soffa 1995] was proposed. A 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 in 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, when the program is executed, can be activated 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 again the program in Figure 1 (a) which contains 4 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 5 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 than in static slicing and the set $\{b_1, \ldots, b_n\}$ are 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 $\langle 17, \{chars\}, \{1, 13, 13, 13, 18\}\rangle$.

```
(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 $\langle 17, \{chars\}, \{1, 13, 13, 13, 18\}\rangle$

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.

It does not mean that the original definition fails to slice multi-procedural 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 (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 traverses procedures up and down. A deep explanation about 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).

```
(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 multi-procedural program

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, Reps and Binkley, 1988)*

In 1988, Horwitz et al. noted that the program dependence graph was not appropriate for representing multi-procedural programs and they proposed a new dependence graph representation of programs called *System Dependence Graph* (SDG) [Horwitz et al. 1988]. This new representation incorporates collections of procedures with procedure calls, and it allows us to produce more precise slices from multi-procedural 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.

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

The SDG of Figure 9 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.

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 stan-

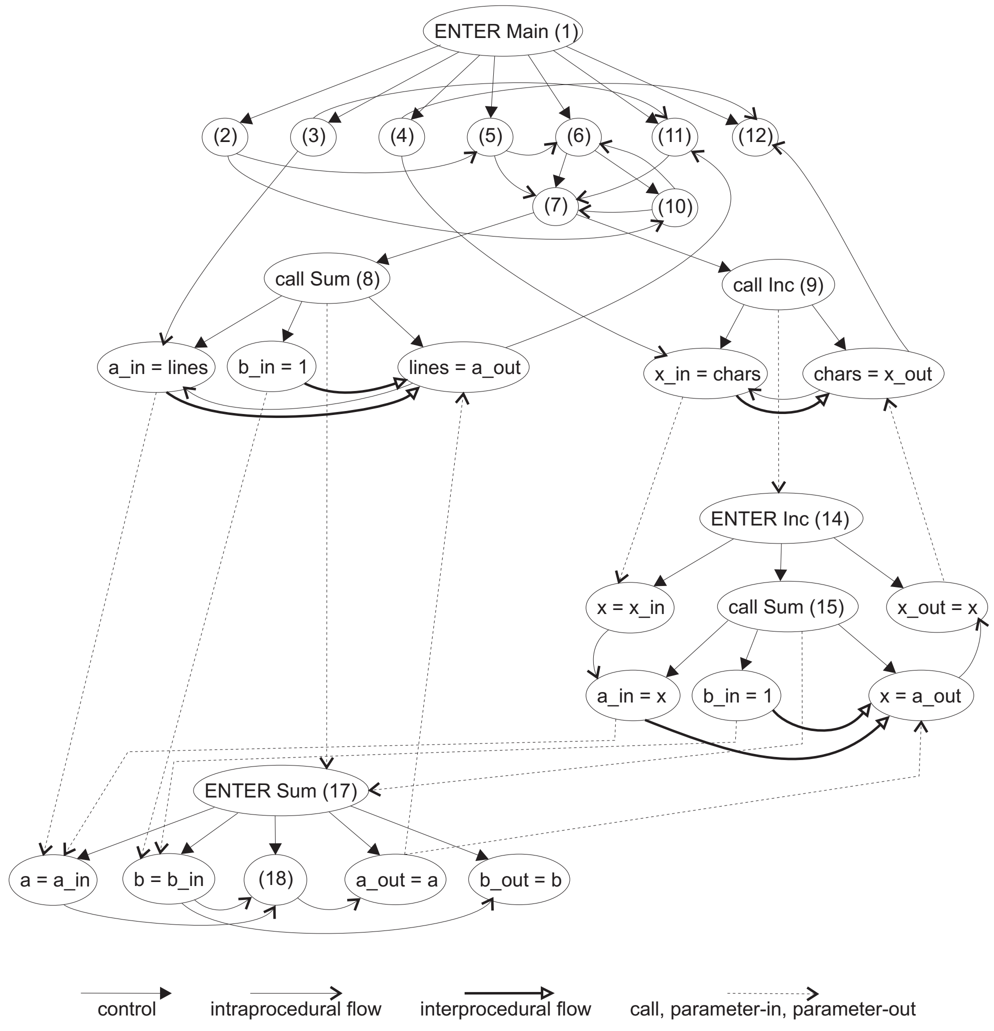


Fig. 10. System Dependence Graph of the program in Figure 9

dard 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 dependences in a class without knowledge of calling environments. ClDGs can be reused in 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. Simi-

```
(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 $\langle 16, \{x\} \rangle$

larly, 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`:

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

Fig. 12. Object-oriented code

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 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 (e.g., in program understanding) interesting to produce a slice with respect to a particular set of executions. 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.

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

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

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, Jihira, Kusumoto and Inoue, 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 counterparts. The reason is 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 be built 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 $\{a_1, \ldots, a_n\}$ are 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 affects 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.

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

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

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, Ohata and Inoue, 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 the 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: First, (i) a standard PDG is constructed without adding data dependence edges; and then, (ii) a data dependence 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 $text$ = "*hello world*!\eof", $n = 4$ 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 (10)→(17); because statements (9) and (10) are never executed. Therefore, if we use the slicing criterion ⟨16, {*lines*}, {*text* = "*hello world*!\eof", $n = 4$}⟩ we get the dependence-cache slice of Figure 4. Note that, in this case, deleting those edges from the PDG is enough as 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, i.e., *text* = "\eof", $n = 0$. 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].

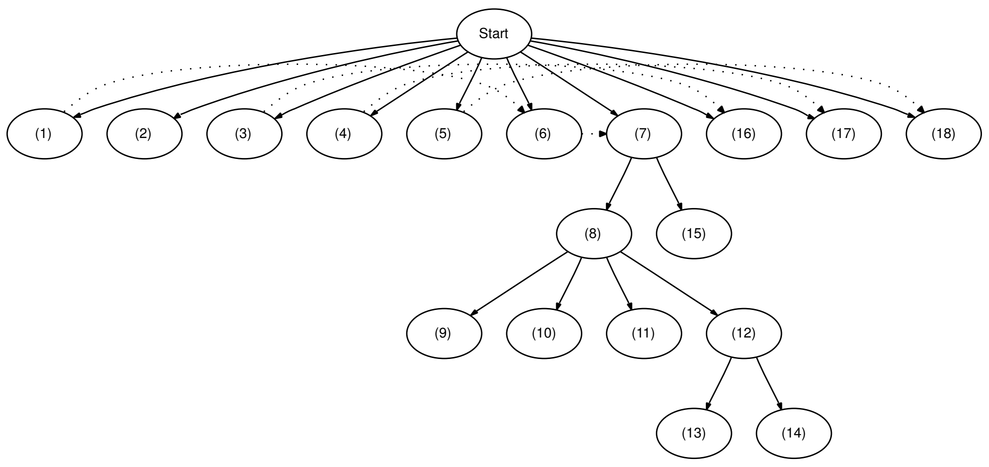


Fig. 15. PDG$_{DS}$ of the program in Figure 1 (a) constructed with respect to the input data *text* = "\eof", $n = 0$

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 for simultaneous slicing. On the one hand, the first definition was *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 reduces to the union of slices (this is not sound) and it requires the use of more elaborated 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 being the rest unknown, in simultaneous dynamic slicing [Hall 1995] a set of "*complete*" inputs is 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 occurrence 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 $(s_i, v_i), 1 \leq n$, 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 unioning the static slices for $\langle 9, lines\rangle$ and $\langle 16, lines\rangle$. However, despite 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

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

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).

In 1993, 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, she usually uses only a part of it. The rest of unused components become dead code. Therefore, the basic idea of interface slicing 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) she is interested in. Then, an interface slicer produces from the original module a new 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 where 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

```
(1) module operations

(2)  procedure line-char-count()
(3)  read(text);
(4)  lines = 1;
(5)  chars = 1;
(6)  c = getChar(text);
(7)  while (c != '\eof')
(8)       if (c == '\n')
(9)       then sum(lines,1);
(10)      else increment(chars);
(11)      c = getChar(text);
(12) write(lines);
(13) write(chars);
(14) return

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

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

(21) procedure decrement(x)
(22) rem(x,1);
(23) return

(24) procedure rem(a, b)
(25) a = a - b;
(26) return
```

Fig. 16. Module which exports five procedures

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 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 do I need to reuse these procedures?

Main applications: Reverse engineering and code reuse.

2.16 Program Dicing *(Lile and Weiser, 1987)*

Program dicing [Lyle and Weiser 1987] was originally defined as: "*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, 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, 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 they do not influence these other variables?

Main applications: Debugging.

2.17 Stop-List Slicing *(Gallagher, Binkley and Harman, 2006)*

Stop-list slicing [Gallagher et al. 2006] is a slicing technique similar to dicing; because both of them use variables on which the programmer is not interested 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 from the slice those variables which are of no interest.

Stop-list slicing augments the slicing criterion with a new list of variables: the set of variables that are 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* dependences starting from them. Note that control dependences 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.

```
(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 $\langle 16, \{lines\}, \{c\} \rangle$

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 where 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 she is 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, Engberts and Kozaczynski, 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; Canfora et al. 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 allows us to specify relations between input values. For instance, condition F above specifies that if `text` is a single line, then `n` must be higher than 0.

```
(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 $F = (\forall c \in text,\ c \neq \text{`}\backslash n\text{'}\ .\ n > 0)$

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 $F = (\forall c \in text, c \neq \text{`}\backslash n)$ 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, Harman, Hierons and Danicic, 2001)*

The definition of conditioned slicing defined by De Lucia et al. has been later referred to as **forward conditioning slicing** because the specified condition affects the initial state—the input—, and this condition is used forwards 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 it 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 its generalization to consider a set of conditions instead of one. As a result, the 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 $\langle\{(\{lines\}, 16), (lines = 1, 16)\}\rangle$.

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

Fig. 19. Backward conditioning slice of Figure 1 (a) w.r.t. $\langle\{(\{lines\}, 16), (lines = 1, 16)\}\rangle$

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 "(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/Post Conditioned Slicing *(Harman, Hierons, Fox, Danicic and Howroyd, 2001)*

Pre/post conditioned slicing [Harman et al. 2001] is a generalized form of conditioned slicing which combines forward and backward conditioning; therefore, it uses at the same time both the forward conditions (called preconditions) of forward conditioning slicing and the backward conditions (called postconditions when negated) of backward conditioning slicing. In pre/post conditioned slicing slices are constructed by removing all the statements except those which are in the execution paths determined by the precondition and that 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 a 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/post conditioned slicing criterion can use at the same time forward and backward conditions; moreover,

```
(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 $\langle 9, \{1, 2, 3, 4, 5, 6, 7, 8, 9\}\rangle$

these condition can be more that two if we allow to insert conditions 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/post conditioned 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 $\{(\{lines\}, 16), (\downarrow \lceil \text{`}\backslash n\text{'} \in text \rceil, 1), (\uparrow \lceil lines <= 1 \rceil, 18)\}$ which would produce the empty slice to denote that whenever variable *text* contains a '$\backslash n$' 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.

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 questions:

(1) *given an execution path, which statements can possibly influence reachability of a given statement s?*,

(2) *why this path is never executed?*, and

(3) *why my program never executes 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/post conditioned 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 all states that can execute the path slice, cause the program to enter in 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.

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

(a) Example program

```
(1)  read(x);
(2)  if (x>0)
(3)     y=1;

(5)  if (x>0)
(6)     if (y=0)
(7)        write("ERROR")
```

(b) Path slice

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

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 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 a this statement?

Main applications: Debugging and testing.

2.23 Abstract Slicing *(Hong, Lee and Sokolsky, 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* do 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 under the statement affects (or is 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 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 $y > x$.

```
(1)  read(x);
(2)  read(y);
(3)  max = x;
(4)  if (y > x)
(5)  then max = y;
(6)  write(max);
```

(a) Example program

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

(b) Abstract slice

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

In our running example, we can produce a slice with respect to the slicing criterion $\langle 16, c =\text{'}\backslash n\text{'}, [(1), text = \text{"}hello\ world!\backslash eof\text{"}]\rangle$. This criterion uses the same dynamic input 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 if the condition $c =$'$\backslash n$' will be satisfied or not.

```
(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 $\langle 16, c =$'$\backslash n$'$, [(1), text =$ "*hello world!*$\backslash$*eof*"$]\rangle$

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", i.e., 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 which 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.

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 the syntax 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`.

```
(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

It is known [Harman and Danicic 1997] that amorphous static slicing subsumes traditional static slicing, i.e., 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 [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 (the one 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 $\langle (text), (text = \text{“}hello\backslash nworld!\backslash n\ \backslash eof\text{”}), 13, \{lines\}\rangle$.

```
(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 $\langle (text), (text = \text{“}hello\backslash nworld!\backslash n\ \backslash eof\text{”}), 13, \{lines\}\rangle$

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 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 [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 they 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

—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.

2.27 Incremental Slicing *(Orso, Sinha and Harrold, 2001)*

Incremental slicing [Orso et al. 2001b] is based on the idea that not all data dependences are equal. In particular, Orso et al. [2001a] distinguish between 24 different types of data dependences. 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.

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

        Program 1                   Program 2                    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`)

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 can be also 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 dependences.

The objective of this technique is to allow the user to focus on a particular type of data dependence. This can be useful, e.g., for program comprehension where the user can initially ignore "weak" data dependences and concentrate on "strong" data dependences. Then, she can incrementally incorporate weaker data dependences in the slice. 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 be affected by, the values of the variables in v at s through transitive control or specified types of data dependences.

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 dependences, 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 dependences 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 dependences 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).

Clearly, incremental slicing generalizes static slicing by only using a subset of data dependences 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 dependences*,
(2) *base pointer flow dependences* (a base pointer flow dependences is a flow dependence due solely to the use of a pointer in a field dereference), and
(3) *producer flow dependences* (those flow dependences which are not base pointer flow dependences).

A thin slice is computed by ignoring control and base pointer flow dependences, 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.

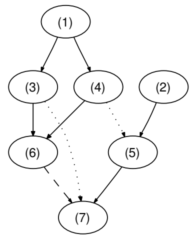
As an example, consider the Java code in Figure 27.

```
(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

In the PDG, thick arrows represent producer flow dependences, dotted arrows represent base pointer flow dependences and dashed arrows represent control dependences. Therefore, following producer flow dependences 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 de 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 are the variables in ψ,

—for each statement appearing in ψ (in the example, (16)),

—for each predecessor of the statements appearing in ψ (in the example, (7), see the CFG in Figure 2),

—for each successor of the statements appearing in ψ (in the example, (17)), and

—for each statements which assigns a value to the variables in ψ (in the example, (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 are needed to satisfy a given LTL formula?

Main applications: Model checking.

2.29 Database Slicing *(Sivagurunathan, Harman and Danicic, 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.

In 1997, Sivagurunathan et al. [1997] noted that standard algorithms produce incorrect slices in presence of I/O and database operations. The reason is that program slicers only consider the program state and they 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 assume 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 dependences which must be computed and added to the PDG—The resultant PDG is known as *Database-Oriented Program Dependence Graph* (DOPDG)—. The new dependences are program-database dependences which relate non-database statements with database statements; and database-database dependences 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. Firstly, the slicing criteria are thoroughly compared by examining their differences for every slicing criteria's dimensions. And, secondly, the techniques are classified with respect to a set of slicing relations which allow 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 concretion, we need first 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 along this section 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 iff $\forall\, n, n \in \mathcal{S}\ .\ n \rightarrow^* s$. 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 say 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;
```

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 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 a particular case of another technique in Section 2 and thus, they do not impose additional restrictions or dimensions in the table. For instance, we omit in the table "End Slicing" 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 in the table the most general form of slicing for each technique. For instance, in the case of quasi-static slicing, the original definition by Venkatesh 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 dependences. For each slicing technique (first column),

—column *Variables* 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* 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* 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* indicates if the slice produced is path-aware as in Korel and Laski style of slicing (see Section 2.3) or it is path-unaware,

—column *Direction* states the direction of the traversal to produce the slice which can be backwards (B), forwards (F) or a combination of them. Here, despite some techniques accept both directions, we assign to every technique the direction specified in its original definition (see Section 2),

—column *Precision* is marked if the slice produced is precise,

—column *Limits* 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. Finally,

—column *Syntax/Semantics Preserving* 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* specifies how many types of dependences are considered to compute the slice.

—column *Statements considered* specifies what statements could be part of the slice. Here, "+" indicates that a subset of statements are included in the slice indeed if they do not contribute to the slicing criterion. Whereas "−" indicates that a subset of statements are not included in the slice indeed 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 which is path-aware; the second one is the path-unaware definition introduced

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

Table I. Classification of program slicing techniques

by Agrawal and Horgan. We put attention on Agrawal and Horgan's definition because it is more extended in the program slicing community and, thus, it is comparable to most slicing techniques. In the case of simultaneous slicing, for concretion, 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. In consequence, 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 that 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 are the eight techniques shown in Table II.

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

Table II. Binkley et al.'s slicing techniques produced by permutation of slicing dimensions

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

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

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 it happens in path slicing, the variables of interests 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 wether 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.

```
(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 the statement 9

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

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 cannot be answered by previous techniques.

As statement slicing, point slicing does consider 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 have they 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 dependences, but control flows, and 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/post conditions of pre/post conditioned 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 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/post conditioned 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 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, they affect sink when some pre and post conditions 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 appears to be correctly computed. However, in the slice still remain many statements which the programmer knows that are correct (e.g., reused procedures and functions, legacy code, etc.). The barriers facility of barrier slicing comes in handy at this point, because they can be set at strategic points (e.g, procedure calls) to avoid the 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 dependences and filtered slicing only considers a type of statements. 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 fulfil 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 $lines$ does 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, c specifies a condition that statements must fulfil in order to be part of the slice and f specifies a condition that statements must fulfil 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 syntaxis preservation 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/post conditions would increase the power of this technique.

3.2 Inter-relations 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 iff all the slicing criteria that can be specified with S_1 can also be specified with S_2. This relation answer 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 iff all the statements in S_1 also belong to S_2. This relation answer 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 iff 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), dicing is composed of n slicing techniques (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 answer 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 parametriza-

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

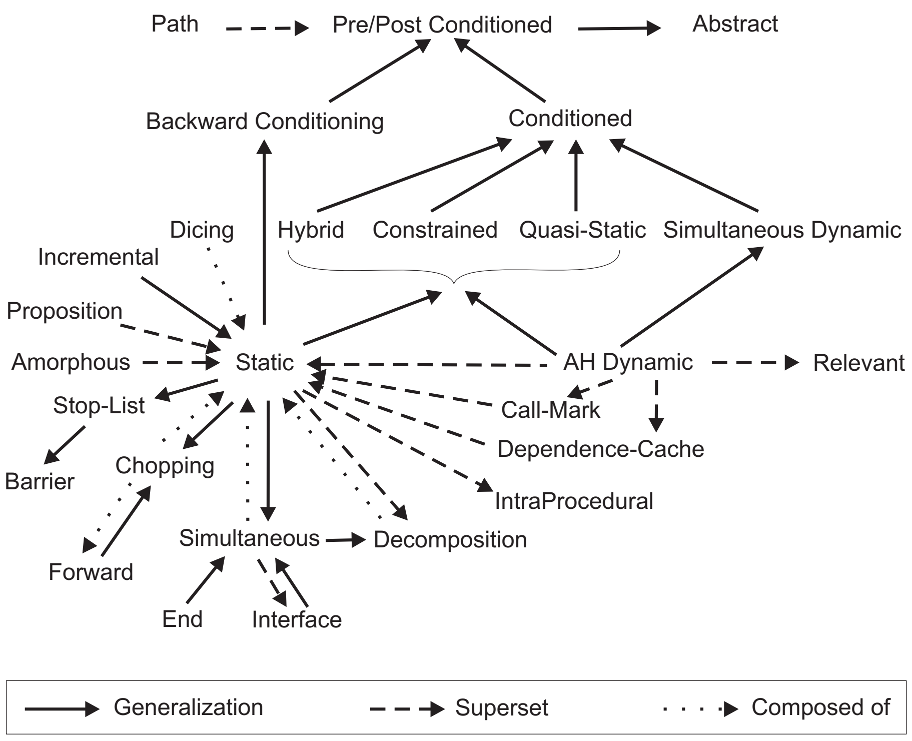


Fig. 29. Relationships between program slicing techniques

tion 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 in 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 31. 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 decade.

We can also focus on a set of dimensions and get interesting conclusions. This is what 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 32. Since each pair of values taken

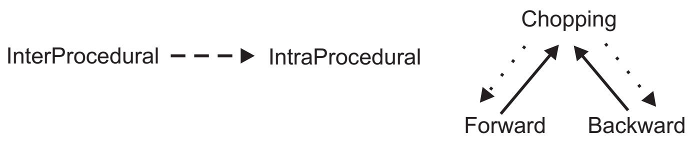


Fig. 30. Relationships between interprocedural and intraprocedural slicing, and between chopping and forward and backward slicing

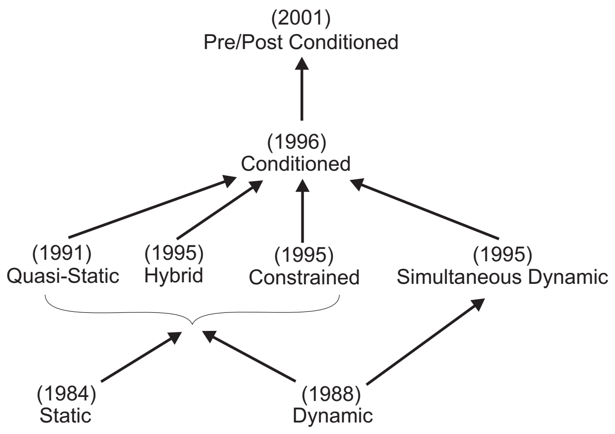


Fig. 31. Generalization relationships between program slicing techniques which develop the *initial states* dimension

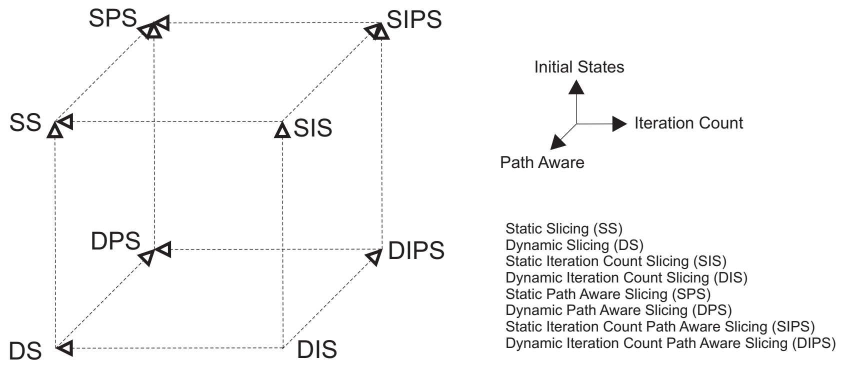

SPS
SIPS
SS
SIS
DPS
DIPS
DS
DIS
Initial States
Iteration Count
Path Aware
Static Slicing (SS)
Dynamic Slicing (DS)
Static Iteration Count Slicing (SIS)
Dynamic Iteration Count Slicing (DIS)
Static Path Aware Slicing (SPS)
Dynamic Path Aware Slicing (DPS)
Static Iteration Count Path Aware Slicing (SIPS)
Dynamic Iteration Count Path Aware Slicing (DIPS)

Fig. 32. Superset relationships between the program slicing techniques of Table II

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.

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. with new dimensions and techniques. The classification of techniques presented not only shows the differences between them, but it also allows us to predict future techniques that will fit a combination of parameters in the table not used yet. 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 inter-relations 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?

REFERENCES

Agrawal, H. 1991. Towards automatic debugging of computer programs. Tech. Rep. Ph.D. Thesis, Purdue University.

Agrawal, H. and Horgan, J. R. 1990. Dynamic program slicing. In *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, Washington, DC, USA, 348–357.

Beck, J. 1993. Interface slicing: a static program analysis tool for software engineering. Ph.D. thesis, Morgantown, WV, USA.

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 Transactions on Programming Languages and Systems 7,* 1, 37–61.

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

Binkley, D., Danicic, S., Gyimóthy, T., Harman, M., Kiss, Á., and Korel, B. 2006. A formalisation 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. 2006. 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 Source Code Analysis* *and Manipulation, Fourth IEEE International Workshop*. IEEE Computer Society, Washington, DC, USA, 43–52.

Binkley, D. and Gallagher, K. B. 1996. Program slicing. *Advances in Computers 43*, 1–50.

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

Canfora, G., Cimitile, A., and De Lucia, A. 1998. Conditioned program slicing. *Information and Software Technology 40,* 11-12 (Nov.), 595–608. Special issue on program slicing.

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 , Victoria, BC, Canada, 1994*. 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 Engineering Bulletin 30,* 4, 22–28.

Cheng, J. 1993. Slicing concurrent programs - a graph-theoretical approach. In *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. *Computing Journal 48,* 6, 737–748.

Danicic, S. and Harman, M. 1996. Program slicing using functional networks. In *Proceedings of 2nd UK Workshop on Program Comprehension (Kyoto University, Japan).* 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* (Florence, Italy). IEEE Computer Society Press, Los Alamitos, California, USA, 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, 26-28 March 2003, Benevento, Italy.* 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 Transactions on Programming Languages and Systems 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 specialisation 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 Transactions on Software Engineering 17,* 8 (Aug.), 751–761.

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

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

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

Gifforn, D. and Hammer, C. 2007. An evaluation of slicing algorithms for concurrent programs. In *Proc. of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE Computer Society, Washington, DC, USA, 17–26.

Greibach, S. 1985. *Theory of program structures: schemes, semantics, verification*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

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 *ESEC/FSE-7: Proceedings of the 7th European Software Engineering Conference*. Springer-Verlag, London, UK, 303–321.

Hall, R. 1995. Automatic extraction of executable program subsets by simultaneous dynamic program slicing. *Automatic Software Engineering 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 UK Workshop on Program Comprehension (Durham University, UK, July 1996)*.

Harman, M. and Gallagher, K. 1998. Program slicing. *Information and Software Technology 40,* 11-12 (Nov.), 577–582. Special issue on program slicing.

Harman, M. and Hierons, R. 2001. An overview of program slicing. *Software Focus 2,* 3, 85–92.

Harman, M., Hierons, R. M., Fox, C., Danicic, S., and Howroyd, J. 2001. Pre/post conditioned 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 Press SIGPLAN Notices, volume 23 (7), Atlanta, GA, 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 2005 ACM SIGPLAN conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 38–47.

Jones, N. 1996. An introduction to partial evaluation. *ACM Computing Surveys 28,* 3 (Sept.), 480–503.

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

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

Krinke, J. 2003. Barrier slicing and chopping. In *Proceedings of the 3rd IEEE International Workshop on Source Code Analysis and Manipulation, 26-27 September 2003, Amsterdam, The Netherlands*. IEEE Computer Society, 81–87.

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

Krinke, J. 2004. Slicing, chopping, and path conditions with barriers. *Software Quality Journal 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, USA, 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, Washington, DC, USA, 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, USA, 647–656.

NANDA, M. AND RAMESH, S. 2000. Slicing concurrent programs. In *Proc. of the ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, New York, NY, USA, 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*. Peking, China, 877–882.

NING, J., ENGBERTS, A., AND KOZACZYNSKI, W. 1994. Automated support for legacy code understanding. *Communications 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 2001)*. Toronto, Canada, 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 2001)*. Firenze, Italy, 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, USA, 177–184.

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

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

SIVAGURUNATHAN, Y., HARMAN, M., AND DANICIC, S. 1997. Slicing, I/O and the implicit state. In *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 LNCS 1292, 291–308.

SRIDHARAN, M., FINK, S., AND BODIK, R. 2007. Thin slicing. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, NY, USA, 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 Software 15,* 2, 105–112.

TIP, F. 1995. A survey of program slicing techniques. *Journal of Programming Languages 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, USA, 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, 26-29 August 2002, Oxford, England*. 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, 26-27 September 2003, Amsterdam, The Netherlands.* IEEE Computer Society, 88–97.

WARD, M. AND ZEDAN, H. 2007. Slicing as a program transformation. *ACM Transactions in Programming Languages and Systems 29,* 2.

WEISER, M. 1979. Program slices: formal, psychological, and practical investigations of an automatic program abstraction method. Ph.D. thesis, The 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 Transactions on Software Engineering 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, Washington, DC, USA, 448–452.

XU, B., QIAN, J., ZHANG, X., WU, Z., AND CHEN, L. 2005. A brief survey of program slicing. *SIGSOFT Software Engineering 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, Washington, DC, USA, 251.

PART II

Algorithmic Debugging Techniques

1. INTRODUCTION

In 1982, Ehud Saphiro presented in his doctoral dissertation [Shapiro 1982] a new debugging technique called *algorithmic debugging*. Since then, algorithmic debugging has been extended and applied to practically all programming paradigms, and different implementations have evolved to produce mature debuggers for several languages. This article surveys more than two decades of work on algorithmic debugging, mainly focussing on the different approaches to improve the performance of algorithmic debugging. It has been written in a way that can be understood by non-experts in algorithmic debugging. In particular, the article starts with an introduction to algorithmic debugging which is explained in a paradigm-independent way through the use of different examples. Then, each algorithmic debugging-based technique contained in the article is studied in a different section. In this way, algorithmic debugging experts can concentrate just on the discussions of the techniques they are interested in. All the techniques are explained with a common running example. This allows us to easily compare one technique with the others by observing the differences they produce with the same example and bug.

The main contributions can be summarized in four items; two of them focus on theoretical studies, and the other two are the result of empirical experimentation:

—A functional requirements specification for algorithmic debugger implementors

—A comparative study of algorithmic debugging strategies

—An empirical evaluation of algorithmic debugging strategies

—A comparative of algorithmic debugging implementations

The first part of this work studies the functional requirements that a modern algorithmic debugger must fulfill. This requirements specification is the result of more than twenty five years of techniques, methods, optimizations, and implementations. By studying the different branches of research which has improved the original Shapiro's technique, we collect the more general technique for each branch and we identify different—independent—ways to improve the performance of an algorithmic debugger.

One of the most important lines of research in algorithmic debugging is the strategy used to traverse the internal data structure used (see Section 2). The second main contribution of this article is an in-depth study of all strategies that has appeared in the literature so far. In this study, for the sake of clarity, we will use a common example, and will produce an algorithmic debugging session to debug the same bug with each strategy. This will illustrate the main differences between the strategies, and their strong and weak points. To conclude this study, we analyze a table which compares the costs of each strategy; and we propose a method to combine different strategies.

The third main contribution is the empirical evaluation of the algorithmic debugging strategies studied. The theoretical study of costs of the algorithmic debugging strategies produces the bounds of these costs (that is the cost of the worst and best cases for each strategy). To study the average costs of the strategies in practice we have evaluated all the strategies with a set of benchmarks.

Finally, the fourth contribution of this article is the empirical comparison of current real implementations. We have collected all the implementations of declarative debuggers (obviously, for different paradigms and languages), and we have evaluated them. The evaluation consisted in measuring their level of maturity by checking how many, and to which degree, do they fulfill the requirements specification presented. Moreover, in a second step, we evaluate the performance of the tools by measuring their time and resources consumption.

This part has been structured as follows: First, we introduce algorithmic debugging and the nomenclature that will be used along the rest of the article. Next, we describe the functional requirements specification. One of the requirements is related to the strategy used during the debugging process. In this point, we will explain all the strategies with a running example, and we will compare and evaluate them. In Section 4 we will evaluate current algorithmic debuggers in order to determine their level of maturity and their performance. In particular, in Section 4.2 we compare the debuggers by studying which features are implemented by which debuggers. This gives rise to a functionality comparison. In Section 4.3 we compare the debuggers from a different perspective. Our aim here is to study the efficiency of the debuggers by comparing their resources consumption. Finally, Section 5 concludes.

2. ALGORITHMIC DEBUGGING

Algorithmic debugging (also called declarative debugging) is a semi-automatic debugging technique which is based on the answers of an oracle (typically the programmer) to a series of questions generated automatically by the algorithmic debugger. These answers provide the debugger with information about the correctness of some (sub)computations of a given program; and the debugger uses them to guide the search for the bug until the portion of code responsible for the buggy behavior is isolated.

Example 2.1. Consider this simple program which wrongly (it has a bug) implements the sorting algorithm *Insertion Sort*:

```
main = insort [2,1,3]

insort [] = []
insort (x:xs) = insert x (insort xs)

insert x [] = [x]
insert x (y:ys) = if x>=y then (x:y:ys)
                          else (y:(insert x ys))
```

An algorithmic debugging session for this program is the following:

```
Starting Debugging Session...
(1)  main = [2,3,1]? NO
(2)  insort [2,1,3] = [2,3,1]? NO
(3)  insort [1,3] = [3,1]? NO
(4)  insort [3] = [3]? YES
(5)  insert 1 [3] = [3,1]? NO
(6)  insert 1 [] = [1]? YES

Bug found in rule:
```

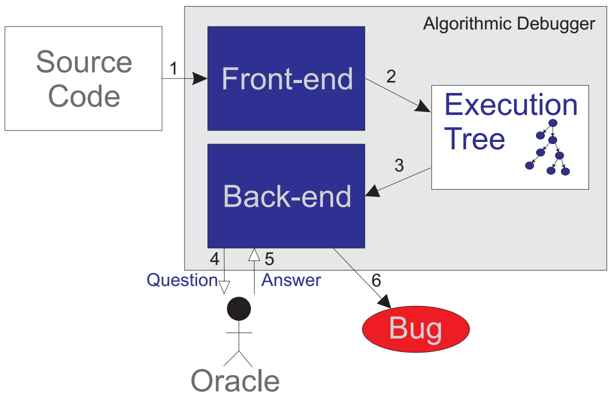


Fig. 33. Architecture of an algorithmic debugger

```
insert x (y:ys) = if x<=y then (x:y:ys)
                             else (y:(insert x ys))
```

The debugger points out the part of the code which contains the bug. In this case, `x<=y` should be `x>=y`. Note that, to debug the program, the programmer only have to answer questions. It is not even necessary to see the code.

Typically, algorithmic debuggers have a front-end which produces a data structure representing a program execution—the so-called *execution tree* (ET)[6] [Nilsson 1998]—; and a back-end which uses the ET to ask questions and process the oracle's answers to locate the bug.

In Figure 33 we can observe the architecture of an algorithmic debugger. Clearly, the schema is very similar to the architecture of a compiler where (i) only the front-end handles the source code; (ii) the front-end produces an intermediate data structure (the ET) which is the input of the back-end; and (iii) the final result is only produced by the back-end. These properties allows one to use the same back-end to debug different languages by only replacing the front-end. Sometimes, the front-end and the back-end are not independent (e.g., in Buddha [Pope 2006] where the ET is generated to main memory when executing the back-end), or they are intertwiningly executed (e.g., in Freja [Nilsson 1998] where the ET is built lazily while the back-end is running).

Depending on the programming paradigm used (i.e., logic, functional, imperative...) the nodes of the ET contain different information: an atom or predicate that was proved in the computation (logic paradigm); or an equation which consists of a function call (functional or imperative paradigm), procedure call (imperative paradigm) or method invocation (object-oriented paradigm) with completely evaluated arguments and results[7], etc. The nodes can also contain additional information about the context of the question. For instance, they can contain constraints (constraint paradigm), attributes values (object-oriented paradigm),

[6]Depending on the programming paradigm, the execution tree is called differently, e.g., *Proof Tree*, *Computation Tree*, *Evaluation Dependence Tree*, etc. We use ET to refer to any of them.
[7]Or as much as needed if we consider a lazy language.

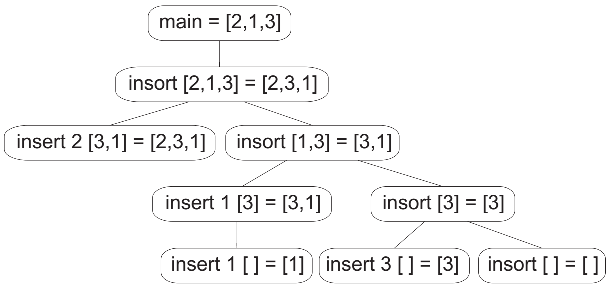


Fig. 34. ET of the program in Example 2.1

or global variables values (imperative paradigm). As we want to study debuggers from heterogeneous paradigms—in order to be general enough—we will simply refer to all the information in an ET's node as *question*, and will avoid the atom/predicate/function/method/procedure distinction unless necessary.

Essentially, algorithmic debugging is a two-phase process: During the first phase, the ET is built. For instance, in the functional paradigm, the ET is constructed as follows: The root node is the *main* function of the program; for each node n with associated function f, and for each function call in the right-hand side of the definition of f which has been evaluated, a new node is recursively added to the ET as the child of n. As an example, the ET of the program in Example 2.1 is depicted in Figure 2.

This notion of ET is valid for functional languages but it is insufficient for other paradigms such as the imperative programming paradigm. In general, the information included in the nodes of the ET includes all the data needed to answer the questions. For instance, in the imperative programming paradigm, with the function (or procedure) of each node it is included the value of all global variables when the function was called. Similarly, in object-oriented languages, every node with a method invocation includes the values of the attributes of the object owner of this method (see, e.g., [Caballero 2006]).

In order to formalize a notion of execution tree that can be used in any paradigm, we need first a definition of software procedure general enough as to be applicable to any paradigm.

Definition 2.2 *Software Procedure.* Given a program $\mathcal{P}$, a software procedure is a part of $\mathcal{P}$ which can be uniquely identified with a name and that it can be invoked (maybe with some parameters) producing a computation which produces a result.

This definition of software procedure can be applicable in different heterogeneous contexts. Examples of software procedures are the methods of a class (in the object-oriented paradigm), the functions of the functional paradigm, the predicates of the logic paradigm and the functions and procedures of the imperative paradigm.

Definition 2.3 *Information Container.* Given a program $\mathcal{P}$, an information container is a part of $\mathcal{P}$ which can be uniquely identified with a name and that can contain a value.

Intuitively, an information container represents the variables and constants of the program. Examples of information containers are constants, local and global variables, and the attributes of a class.

Definition 2.4 *Scope of Software Procedures.* Let $\mathcal{P}$ be a program, $p \in \mathcal{P}$ a software procedure and $v \in \mathcal{P}$ an information container. We say that v is in the scope of p iff the value of v could be used or changed during the execution of p.

Definition 2.5 *ET Node.* An ET node represents the execution of a software procedure. Hence, an ET node contains the information needed to perform a software procedure call and the effect of this call. Let $\mathcal{P}$ be a program and $p \in \mathcal{P}$ a software procedure. Then, for every call to p during the execution of $\mathcal{P}$, there is an ET node which contains:

—The name of p.
—The names and values of the parameters of the call to p before and after the execution of p.
—The output of p.
—All the information containers of the program together with their values before and after the execution of p, that are in the scope of p except those information containers which have been defined inside p.

Obviously, the information inside an ET node can be represented—and shown to the user—in many different ways. The most common way to represent the information inside the ET's nodes is through the use of equations whose left-hand side represents the call, and whose right-hand side shows the effects of the call (see, e.g., Figure 2).

Example 2.6. Consider the following Java program:

```
Class Circle{
Point center;
float radius;

void Circle(){
   center = (0,0);
   radius = 2;
}

float getArea(){
   float area = PI * (radius^2);
   return area;
}}
```

Consider also a circle class `C` (`Circle C = new Circle();`) and the invocation of methods `C.circle(); C.getArea()`. Then, the ET nodes that respectively represent the execution of `C.Circle()` and `C.getArea()` contain:

```
{C.center = (0,0), C.radius = 2}    {C.center = ⊥, C.radius = ⊥}
C.getArea() -> 12.57                C.circle() -> ⊥
{C.center = (0,0), C.radius = 2}    {C.center = (0,0), C.radius = 2}
```

where $\perp$ represents a NULL value.

Definition 2.7 *Execution Tree.* Given a program $\mathcal{P}$ with software procedures $p_i \in \mathcal{P}, 0 \leq i \leq n$, and a call c to p_i, the execution tree of $\mathcal{P}$ w.r.t c is a tree whose nodes are ET nodes and

—The root of the ET is the ET node associated to c

—For each ET node α, associated to a call c' to $p_j, 0 \leq j \leq n$, we have a child node β, associated to a call c'' to $p_k, 0 \leq k \leq n$, iff

(1) during the execution of c', c'' is invoqued, and

(2) the call c'' is done from the definition of p_j.

Once the ET is built, in the second phase, the debugger uses a strategy to traverse the ET asking an oracle (typically the programmer) whether each equation is correct or wrong. At the beginning, the *suspicious area* which contains those nodes that can be buggy (a buggy node is associated with the buggy software procedure of the program) is the whole ET; but, after every answer, some nodes of the ET leave the suspicious area. When all the children of a node with a wrong equation (if any) are correct, the node becomes buggy and the debugger locates the bug in the part of the program associated to this node [Nilsson and Fritzson 1994]. If a bug symptom is detected then algorithmic debugging is complete [Shapiro 1982]. It is important to say that, once the ET is built, the problem of traversing it and selecting a node is mostly independent of the language used (a clear exception are those strategies which use information about the syntax of the program); hence many algorithmic debugging strategies can theoretically work for any language.

Unfortunately, in practice—for real programs—algorithmic debugging can produce long series of questions which are semantically unconnected (i.e., consecutive questions which refer to different and independent parts of the computation) making the process of debugging too complex.

Furthermore, questions can also be very complex. For instance, during a debugging session with a compiler, the algorithmic debugger of the Mercury language [MacLarty 2005] asked a question of more than 1400 lines.

Therefore, efficient techniques and strategies to reduce the number of questions, to simplify them and to improve the order in which they are asked are a necessity to make algorithmic debuggers usable in practice.

As shown in Example 2.1, during the algorithmic debugging process, an oracle is prompted with equations and asked about their correctness; it answers "YES" when the result is correct or "NO" when the result is wrong. Some algorithmic debuggers also accept the answer "I don't know" when the programmer cannot give an answer (e.g., because the question is too complex). After every question, some nodes of the ET leave the suspicious area. When there is only one node in the suspicious area, the process finishes reporting this node as buggy. It should be clear that, given a program with a wrong behavior, this technique guarantees that, whenever the oracle answers all the questions, the bug will eventually be found. However, algorithmic debugging finds one bug at a time. In order to find different bugs, the process should be restarted again for each different bug. Of course, once the first bug is removed, algorithmic debugging may be applied again in order to find another bug; and modern algorithmic debuggers keep a record of previous answers of the oracle to avoid the repetition of questions.

```
main = sqrtest [1,2]

sqrtest x = test (computs (listsum x))

test (x,y,z) = (x==y) && (y==z)

listsum [] = 0
listsum (x:xs) = x + (listsum xs)

computs x = ((comput1 x),(comput2 x),(comput3 x))

comput1 x = square x

square x = x*x

comput2 x = listsum (list x x)

list x y | y==0      = []
         | otherwise = x:list x (y-1)

comput3 x = listsum (partialsums x)

partialsums x = [(sum1 x),(sum2 x)]

sum1 x = div (x * (incr x)) 2
sum2 x = div (x + (decr x)) 2

incr x = x + 1
decr x = x - 1
```

Fig. 35. Example program

Let us illustrate the process with a slightly greater example that we will use along the paper as our running example[8].

Example 2.8. Consider the buggy program in Figure 35 adapted to Haskell from [Fritzson et al. 1992]. This program sums a list of integers [1,2] and computes the square of the result with three different methods. If the three methods compute the same result the program returns *True*; otherwise, it returns *False*. Here, one of the three methods—the one adding the partial sums of its input number—contains a bug. From this program, an algorithmic debugger can automatically generate the ET of Figure 37 (for the time being, the reader can ignore the distinction between different shapes and white and dark nodes) which, in turn, can be used to

[8]While almost all the techniques and algorithmic debugging strategies presented here are independent of the programming paradigm used, in order to be concrete and w.l.o.g. we will base our examples on the functional programming paradigm.

```
Starting Debugging Session...
(1)  main = False? NO
(2)  sqrtest [1,2] = False? NO
(3)  test [9,9,8] = False? YES
(4)  computs 3 = [9,9,8]? NO
(5)  comput1 3 = 9? YES
(7)  comput2 3 = 9? YES
(16) comput3 3 = 8? NO
(17) listsum [6,2] = 8? YES
(20) partialsums 3 = [6,2]? NO
(21) sum1 3 = 6? YES
(23) sum2 3 = 2? NO
(24) decr 3 = 2? YES

Bug found in rule:
sum2 x = div (x + (decr x)) 2
```

Fig. 36. Debugging session for the program in Figure 35

produce a debugging session as depicted in Figure 36. During the debugging session, the system asks the oracle about the correctness of some ET's nodes w.r.t. the intended semantics. At the end of the debugging session, the algorithmic debugger determines that the bug of the program is located in function "`sum2`" (node 23). The definition of function "`sum2`" should be: `sum2 x = div (x*(decr x)) 2`.

3. FUNCTIONAL REQUIREMENTS OF AN ALGORITHMIC DEBUGGER

Since the first implementation, algorithmic debuggers have evolved significantly. Modern implementations of algorithmic debuggers incorporate many new techniques, strategies and facilities that speed up and ease the debugging process. In this section we describe the main techniques and features of modern algorithmic debuggers. All of them together form a functional requirements specification.

3.1 Accepted Answers

Algorithmic debugging strategies are based on the fact that the ET can be pruned using information provided by the oracle. Given a question associated to a node n in the ET, the debugger should be able to accept the following answers from the oracle:

—"Yes" to indicate that the node is correct or valid. In this case, the debugger prunes the subtree rooted at n, and the search continues with the next node according to the selected strategy.

—"No" when the node is wrong or invalid. This answer prunes all the nodes of the ET except the subtree rooted at n, and the search strategy continues with a node in this subtree.

—"I Don't Know" to skip a question when the user cannot answer it (e.g., because it is too difficult).

—"Inadmissible". In 1986, Pereira [1986] noted that the answers "Yes", "No" and "I Don't Know" were insufficient; and he pointed out another possible answer of the programmer: *Inadmissible* (see also [Naish 1997b]). An equation or, more

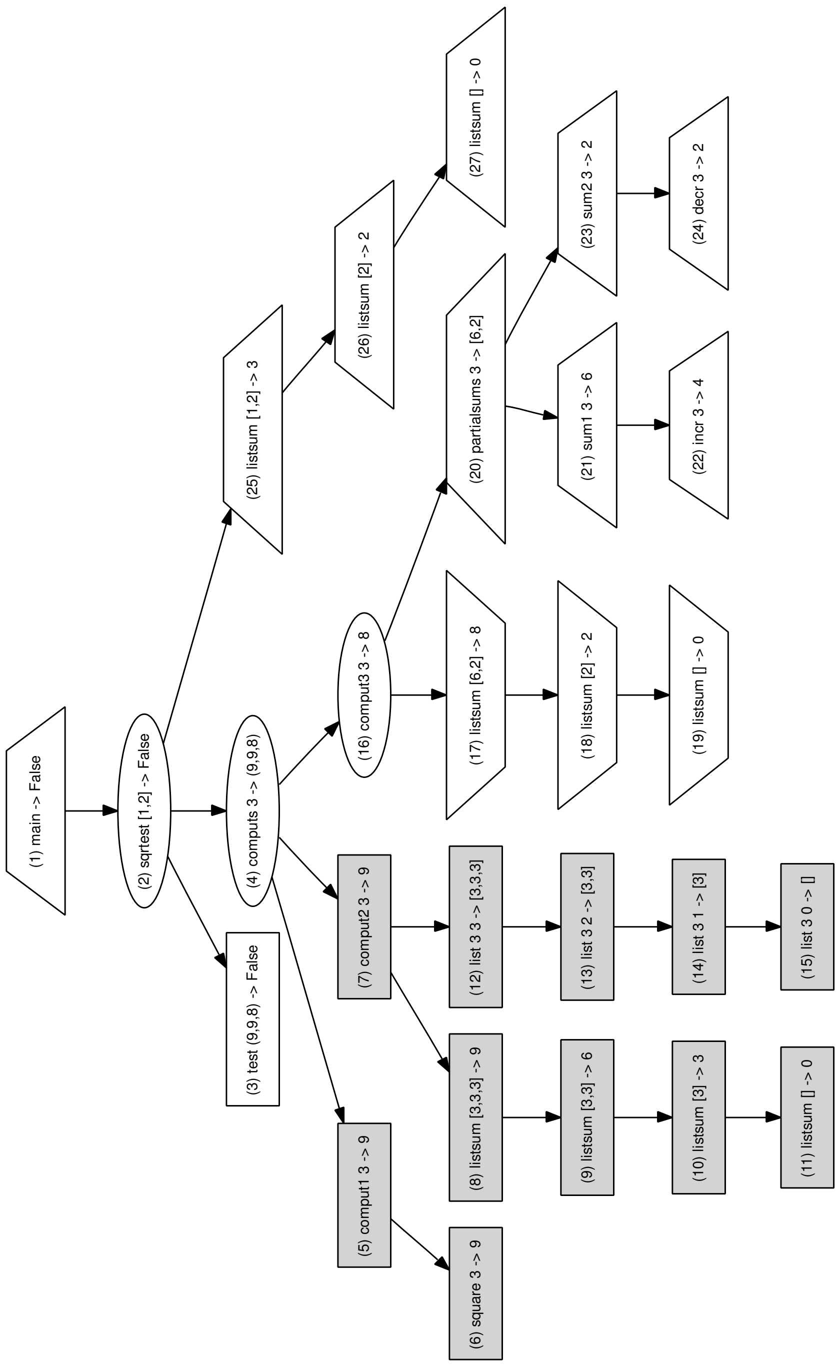


Fig. 37. Execution tree of the program in Figure 35

precisely, some of its arguments, are inadmissible if they violate the preconditions of its function definition.

Example 3.1. Consider the equation

```
insert ‘b’ “cc” = “bcc”
```

where function `insert` inserts the first argument in a list of mutually different characters (the second argument). This equation is not wrong but inadmissible, because the argument “`cc`” has repeated characters. Hence, inadmissibility allows us to identify errors in left-hand sides of equations.

Inadmissibility allows the user to specify that some argument in the atom/predicate or function/method/procedure call associated to the question should not have been computed (i.e., it violates the preconditions of the atom/predicate/function/method/procedure). Answering “Inadmissible” to a question redirects the search for the bug in a new direction related to the nodes which are responsible for the inadmissibility. That is, those nodes that could have influenced the inadmissible argument [Silva and Chitil 2006].

—“Trusted” [?] to indicate that some module, argument or predicate/function/method/procedure in the program is trusted (e.g., because it has been already tested). All the rest of nodes related to the trusted module, argument or predicate/function/method/procedure should also be automatically trusted.

3.2 Tracing Subexpressions

A “No” or an “Inadmissible” answer is often too imprecise. When the programmer knows exactly which part of the question is wrong, she could mark a subexpression as wrong. For instance, in the equation

```
wordcharcount “Hello world!” = (2,24)
```

not all the result is wrong; only the char count is wrong, and thus the debugger should avoid asking questions related to the computation of the word count. Therefore, the answer “No” is too imprecise here, because the system fails to get fundamental information from the programmer about where exactly is the wrong part.

Tracing subexpressions avoids many useless questions which are not related to the wrong subexpression. For instance, Mercury’s debugger [MacLarty 2005] uses subexpression information to enhance bug search. In particular, the algorithmic debugging strategy “Subterm Dependency Tracking” (see Section 3.10.9) bases the search for the bug on subexpressions marked by the oracle.

Tracing subexpressions has two main advantages. First, it reduces the search space because the debugger only explores the part of the ET related to the wrong subexpression. Second, it makes the debugging process more understandable, because it gives the user some control over the bug search.

3.3 Tree Compression

Tree compression is a technique used to remove redundant nodes from the ET [Davie and Chitil 2006].

```
(1) append [] y = y
(2) append (x:xs) y = x:append xs y
```

Fig. 38. Definition of append function.

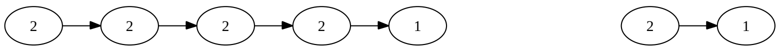


Fig. 39. ET of `append [1,2,3,4] [5,6]` and its associated compressed tree.

Let us illustrate the technique with an example. Consider the function definition in Figure 38, and its associated ET for the call "`append [1,2,3,4] [5,6]`" in Figure 39 (left). The function `append` makes three recursive calls to rule `(2)` and, finally, one call to the base case in rule `(1)`. Clearly, there are only two rules that can be buggy: rules 1 and 2. Therefore, the ET can be compressed as depicted in Figure 39 (right). With this ET, the debugger will only ask at most two questions. Note that with a list of n numbers as the first argument, standard strategies could need $n+1$ questions to reach the bug.

This technique proceeds as follows: For any node n_1 with associated rule r_1 and child n_2 with associated rule r_1, it replaces node n_2 with its children (i.e., the children of n_2 become the children of n_1). Therefore, the tree compression technique prunes ETs even before starting to ask any question, hence, it should be implemented by all the debuggers as a postprocess to the computation of the ET.

3.4 Memoization

The debugger should not ask the same question twice. This can be easily done by memoizing the answers of the programmer. Memoization can be done intra- or inter-session.

Before debugging, programs often contain more than one bug. However, algorithmic debuggers only find one bug with each session. Therefore, in order to find more than one bug, the programmer must execute the debugger several times, and thus, the same questions can be repeated in different sessions.

To avoid this situation, the debugger should store the user's answers and reuse them in future sessions. Hence, in successive debugging sessions of the same program, the same question is not asked once and again.

3.5 Graphical User Interface

A GUI can speed up the debugging session, because it allows the user to freely explore the ET and mark nodes independently of (or in parallel with) the running strategy. Graphical features such as collapsing subcomputations of the ET can be very useful.

3.6 Undo Capabilities

Not only the program can be buggy. Also the debugging session itself could be buggy, e.g., when the programmer answers a question incorrectly. Therefore, the debugger should allow the programmer to correct wrong answers. For instance, a desirable feature is allowing the user to undo the last answer and return to the previous state. Despite it seems to be easy to implement, surprisingly, most algorithmic debuggers lack of an *undo* command; and the programmer is forced to repeat the whole session when she does a mistake with an answer.

Some debuggers drive buggy debugging sessions in a different manner. For instance, Freja uses the answers *maybe yes* and *maybe no* in addition to *yes* and *no*. They are equal to their counterparts except that the debugger will remember that no definitive answer has been given, and return to those questions later unless the bug has been found through other answers first.

3.7 ET Exploration

Algorithmic debugging can become too rigid when it is only limited to questions generation. Sometimes, the programmer has an intuition about the part of the ET where the bug can be. In this situation, allowing the programmer to freely explore the ET could be the best strategy. It is desirable to provide the programmer with the control over the ET exploration when she wants to direct the search for the bug.

3.8 Trusting

Programs usually reuse code already tested (e.g., external functions, modules...). Therefore, when debugging a program, this code should be trusted. Trusting can be done at the level of modules, questions or arguments. And it can be done statically (by including annotations or flags when compiling) or dynamically (the programmer answers a question with "Trusted" and all the questions referring to the same atom/predicate/function/method/procedure are automatically set "Correct").

3.9 Scalability

One of the main problems of current algorithmic debuggers is their low scalability. The ETs produced by real programs can be huge (indeed gigabytes) and thus it does not fit in main memory.

Nevertheless, many debuggers store the ET in main memory; hence, they produce a "memory overflow" exception when applied to real programs. Current solutions include storing the ET in secondary memory (e.g., [Davie and Chitil 2006]) or producing the ET on demand (e.g., [Nilsson 1998; MacLarty 2005]).

A mixture between main and secondary memory would be desirable. It would be interesting to load a cluster of nodes in main memory and explore them until a new cluster is needed. This solution would take advantage of the speed acquired by working on main memory (e.g., keeping the possibility to apply strategies on the loaded cluster) while being able to store huge ETs in secondary memory.

A new approach developed in the debugger B.i.O. changes time by space: It does not need to store an ET because it reexecutes the program once and again to generate the next question. As B.i.O. debugs Curry, which is a lazy language, it firstly executes the program and records a file with step counts specifying how much the subcomputations have been evaluated. This file is later used by the back-end to reexecute the program eagerly once and again in order to produce the questions as they are required.

3.10 Multiple Search Strategies

One of the most important metrics to measure the performance of a debugger is the time spent to find the bug. In the case of algorithmic debuggers it is $q * t$ where q is the number of questions asked and t is the average time spent by the oracle to answer a question [Silva 2007a].

An algorithmic debugging strategy determines how to traverse the ET in order to select the next question to ask. Algorithmic debugging strategies are based on the fact that the ET can be pruned using the information provided by the oracle. Given a question associated with a node n of the ET, a NO answer prunes all the nodes of the ET except the subtree rooted at n; and a YES answer prunes the subtree rooted at n. Each strategy takes advantage of this property in a different manner.

A correct equation in the tree does not guarantee that the subtree rooted at this equation is free of errors. It can be the case that two buggy nodes caused the correct answer by fluke [Davie and Chitil 2006]. In contrast, an incorrect equation does guarantee that the subtree rooted at this equation does contain a buggy node [Naish 1997a]. Therefore, if a program produced a wrong result, then the equation in the root of the ET is wrong and thus there must be at least one buggy node in the ET. We will assume in the following that the debugging session has been started after discovering a bug symptom in the output of the program, and thus the root of the tree contains a wrong equation. Hence, we know that there is at least one bug in the program. We will also assume that the oracle is able to answer all the questions. Then, all the strategies will find the bug.

Different strategies have arisen to minimize both the number of questions and the time needed to answer the questions. Firstly, the number of questions can be reduced by pruning the ET (e.g., the strategy Divide and Query [Shapiro 1982] prunes near half of the ET after every answer). Secondly, the time needed to answer the questions can be reduced by avoiding complex questions, or by producing a series of questions which are semantically related. For instance, the strategy Top-Down Zooming [Maeji and Kanamori 1987] tries to ask questions related to the same recursive (sub)computation.

Therefore, the effectiveness of the debugger is strongly dependent on the number, order, and complexity of the questions asked. Surprisingly, some algorithmic debuggers do not implement more than one strategy thus the programmer is forced to follow a rigid and predefined order of questions.

In the following, we enumerate and describe all the strategies that have appeared in the literature and we compare their cost, and study how to combine them.

3.10.1 *Single Stepping* (Shapiro, 1982).

The first algorithmic debugging strategy to be proposed was *single stepping* [Shapiro 1982]. In essence, this strategy performs a bottom-up search because it proceeds by doing a post-order traversal of the ET. It asks first about all the children of a given node, and then (if they are correct) about the node itself. If the equation of this node is wrong then this is the buggy node; if it is correct, then the post-order traversal continues. Therefore, in this strategy, all the nodes without children (the leaves) are asked first, then their parents, and so on. Hence, the first node answered NO is identified as buggy (because all its children have already been answered YES).

For instance, the sequence of 19 questions asked for the ET in Figure 37 is:

```
Starting Debugging Session...

(3)  test [9,9,8] = False? YES       (7)  comput2 3 = 9? YES
(6)  square 3 = 9? YES               (19) listsum [] = 0? YES
(5)  comput1 3 = 9? YES              (18) listsum [2] = 2? YES
(11) listsum [] = 0? YES             (17) listsum [6,2] = 8? YES
(10) listsum [3] = 3? YES            (22) incr 3 = 4? YES
(9)  listsum [3,3] = 6? YES          (21) sum1 3 = 6? YES
(8)  listsum [3,3,3] = 9? YES        (24) decr 3 = 2? YES
(15) list 3 0 = []? YES              (23) sum2 3 = 2? NO
(14) list 3 1 = [3]? YES
(13) list 3 2 = [3,3]? YES            Bug found in rule:
(12) list 3 3 = [3,3,3]? YES          sum2 x = div (x + (decr x)) 2
```

Note that in this strategy questions are semantically unconnected (i.e., consecutive questions refer to independent parts of the computation).

3.10.2 *Top-Down Search* (Av-Ron, 1984).

Due to the fact that questions are asked in a logical order, *top-down search* [Av-Ron 1984] is the strategy that has been traditionally used (see, e.g., [Caballero 2005; Kokai et al. 1999]) to measure the performance of different debugging tools and methods. It basically consists in a top-down, left-to-right traversal of the ET and, thus, the node asked is always a child or a sibling of the previous question node. When a node is answered NO, one of its children is asked; if it is answered YES, one of its siblings is. Therefore, the idea is to follow the path of wrong equations from the root of the tree to the buggy node.

For instance, the sequence of 12 questions asked for the ET in Figure 37 is shown in Figure 36.

This strategy significantly improves single stepping because it prunes a part of the ET after every answer. However, it is still very naive, since it does not take into account the structure of the tree (e.g., how balanced it is). For this reason, a number of variants aiming at improving it can be found in the literature.

3.10.3 *Top-Down Zooming* (Maeji and Kanamori, 1987).

During the search of previous strategies, the rule or indeed the function definition may change from one query to the next. If the oracle is human, this continuous change of function definitions slows down the answers of the programmer because she has to switch thinking once and again from one function definition to another. This drawback can be partially overcome by changing the order of the questions: In this strategy [Maeji and Kanamori 1987], recursive child calls are preferred.

The sequence of questions asked for the ET in Figure 37 is exactly the same as with top-down search (Figure 36) because no recursive calls are found. A difference between both strategies appears when asking the children of node 12; while top-down search asks node 13 first, top-down zooming asks node 14, thus favoring the recursive call.

Another variant of this strategy called *exception zooming*, introduced by Ian MacLarty [MacLarty 2005], selects first those nodes that produced an exception at runtime.

3.10.4 *Heaviest First* (Binks, 1995).

Selecting always the left-most child does not take into account the size of the subtrees that can be explored. Binks proposed in [Binks 1995] a variant of top-down search in order to consider this information when selecting a child. This variant is called *heaviest first* because it always selects the child with a bigger subtree. The objective is to avoid selecting small subtrees which have a lower probability of containing the bug.

For instance, the sequence of 9 questions asked for the ET in Figure 37 would be[9]:

```
Starting Debugging Session...

(1)  main = False? NO
(2)  sqrtest [1,2] = False? NO
(4)  computs 3 = (9,9,8)? NO
(7)  comput2 3 = 9? YES
(16) comput3 3 = 8? NO
(20) partialsums 3 = [6,2]? NO
(21) sum1 3 = 6? YES
(23) sum2 3 = 2? NO
(24) decr 3 = 2? YES

Bug found in rule:
sum2 x = div (x + (decr x)) 2
```

3.10.5 *Divide & Query* (Shapiro, 1982).

In 1982, together with single stepping, Shapiro proposed another strategy: the so-called divide & query (D&Q) [Shapiro 1982]. The idea of D&Q is to ask in every step a question which divides the remaining nodes in the ET by two, or, if this is not possible, into two parts with a weight as similar as possible. In particular, the original algorithm by Shapiro always chooses the heaviest node whose weight is less than or equal to $w/2$ where w is the weight of the suspicious area in the ET. This strategy has a worst case query complexity of order $b\ log_2\ n$ where b is the average branching factor of the tree and n its number of nodes.

This strategy works well with a large search space—this is normally the case of realistic programs—because its query complexity is proportional to the logarithm of the number of nodes in the tree. If the ET is big and unbalanced this strategy

[9]Here, and in the following, we will break the indeterminism by selecting the left-most node in the figures. For instance, the fourth question could be either `(7)` or `(16)` because both have a weight of 9. We selected `(7)` because it is on the left.

is better than top-down search [Caballero 2005]; however, the main drawback of this strategy is that successive questions may have no connection, from a semantic point of view, with each other. This is due to the fact that the questions asked are often related to different and independent parts of the computation; and thus, the programmer requires more time for answering the questions.

For instance, the sequence of 6 questions asked for the ET in Figure 37 is:

```
Starting Debugging Session...

(7)  comput2 3 = 9? YES
(16) comput3 3 = 8? NO
(17) listsum [6,2] = 8? YES
(21) sum1 3 = 6? YES
(24) decr 3 = 2? YES
(23) sum2 3 = 2? NO

Bug found in rule:
sum2 x = div (x + (decr x)) 2
```

3.10.6 *Hirunkitti's Divide & Query* (Hirunkitti and Hogger, 1993).

In [Hirunkitti and Hogger 1993], Hirunkitti and Hogger noted that Shapiro's algorithm does not always choose the node closest to the halfway point in the tree and addressed this problem slightly modifying the original D&Q algorithm. Their version of D&Q is the same as the one by Shapiro except that their version always chooses a node which produces a least difference between:

—$w/2$ and the heaviest node whose weight is less than or equal to $w/2$

—$w/2$ and the lightest node whose weight is greater than or equal to $w/2$

where w is the weight of the suspicious area in the computation tree.

For instance, the sequence of 6 questions asked for the ET in Figure 37 is:

```
Starting Debugging Session...

(7)  comput2 3 = 9? YES
(16) comput3 3 = 8? NO
(17) listsum [6,2] = 8? YES
(21) sum1 3 = 6? YES
(24) decr 3 = 2? YES
(23) sum2 3 = 2? NO

Bug found in rule:
sum2 x = div (x + (decr x)) 2
```

3.10.7 *Biased Weighting Divide & Query* (MacLarty, 2005).

MacLarty proposed in his PhD thesis [MacLarty 2005] that not all the nodes should be considered equally while dividing the tree. His variant of D&Q divides the tree by only considering some kinds of nodes and/or by associating a different weight to every kind of node.

In particular, his algorithmic debugger was implemented for the functional logic language Mercury [Conway et al. 1995] which distinguishes between 13 different node types.

3.10.8 *Hat Delta* (Davie and Chitil, 2005).

Hat [Wallace et al. 2001] is a tracer for Haskell. Davie and Chitil introduced a declarative debugger tool based on the Hat's traces that includes a new strategy called Hat Delta [Davie and Chitil 2005]. Initially, Hat Delta is identical to top-down search but it becomes different as the number of questions asked increases. The main idea of this strategy is to use previous answers of the oracle in order to compute which node has an associated rule that is more likely to be wrong (e.g., because it has been answered NO more times than the others).

Consider for instance that after 20 questions, rule "A" has been answered 10 times NO, while rule "B" has been answered 10 times YES. With this information, rule "A" is more likely to be wrong than rule "B".

This strategy assumes that a rule answered NO n times out of m is more likely to be wrong than a rule answered NO n' times out of m if $n' < n \leqslant m$. During a debugging session, a sequence of questions, each of them related to a particular rule, is asked. In general, after every question, it is possible to compute the total number of questions asked for each rule, the total number of answers YES/NO, and the total number of nodes associated with this rule. Moreover, when a node is set correct or wrong, Hat Delta marks all the rules of its descendants as correctly or incorrectly executed respectively. This strategy uses all this information to select the next question. In particular, three different heuristics have been proposed based on this idea [Davie and Chitil 2006]:

—*Counting the number of YES answers.* If a rule has been executed correctly before, then it will likely execute correctly again. The debugger associates to each rule of the program the number of times it has been executed in correct computations based on previous answers.

—*Counting the number of NO answers.* This is analogous to the previous heuristic but collecting wrong computations.

—*Calculating the proportion of NO answers.* This is derived from the previous two heuristics. For a node with associated rule r we have:

$$\frac{\textit{number of answers NO for } r}{\textit{number of answers NO/YES for } r}$$

If r has not been asked before a value of $\frac{1}{2}$ is assigned.

Example 3.2. Consider this program:

```
4|0|0     sort [] = []
8|4|1/3   sort (x:xs) = insert x (sort xs)
4|0|0     insert x [] = [x]
          insert x (y:ys)
4|0|0       | x<y = x:y:ys
```

```
0|0|1/2     | otherwise = insert x ys
```

where the left numbers indicate respectively the number of times each rule has been executed correctly, the number of times each rule has failed and the proportion of NO answers for this rule.
With this information, `otherwise = insert x ys` is more likely to be wrong (it has a probability of $\frac{1}{2}$ to be wrong).

3.10.9 *Subterm Dependency Tracking* (MacLarty et al., 2005).

The concept of inadmissibility (see Section 3) allows us to specify that the question itself is wrong. However, it does not allow us to say *why* the equation is wrong or inadmissible. In particular, the programmer could specify which exact (sub)term in the result or the arguments is wrong or inadmissible respectively. This provides specific information about *why* an equation is wrong (i.e., which part of the result is incorrect? is one particular argument inadmissible?).

Consider again the equation `insert 'b'` "`cc`" = "`bcc`" of Example 3.1. Here, the programmer could detect that the second argument should not have been computed; she could then mark the second argument ("`cc`") as inadmissible. This information is essential because it allows the debugger to avoid questions related to the correct parts of the equation and concentrate on the wrong parts.

Based on this idea, MacLarty [2005] proposed a new strategy called subterm dependency tracking. Essentially, once the programmer selects a particular wrong subterm, this strategy searches backwards in the computation for the node that introduced the wrong subterm. All the nodes traversed during the search define a *dependency chain* of nodes between the node that produced the wrong subterm and the node where the programmer identified it. The sequence of questions defined in this strategy follows the dependency chain from the origin of the wrong subterm.

For instance, if the programmer is asked question 3 from the ET in Figure 37, his answer would be YES but she could also mark subexpression "8" as inadmissible. Then, the system would compute the chain of nodes which passed this subexpression from the node which computed it until question 3. This chain is formed by nodes 4, 16 and 17. The system would ask first 17, then 16, and finally 4 following the computed chain.

In our example, the sequence of 8 questions asked for the ET in Figure 37, combining this strategy with top-down search, is:

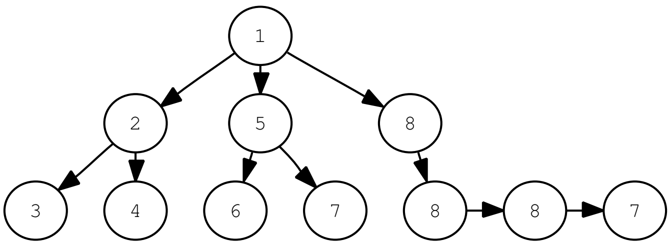
```
Starting Debugging Session...

(1)  main = False? NO
(2)  sqrtest [1,2] = False? NO
(3)  test (9,9,8) = False? YES  (the programmer marks ''8")
(17) listsum [6,2] = 8? YES
(16) comput3 3 = 8? NO
(20) partialsums 3 = [6,2]? NO  (the programmer marks ''2")
(23) sum2 3 = 2? NO
(24) decr 3 = 2? YES
```

```
Bug found in rule:
sum2 x = div (x + (decr x)) 2
```

3.10.10 *Less YES First* (Silva, 2006).

Less YES First [Silva 2006] is a variant of top-down search which further improves heaviest first. It is based on the fact that every equation in the ET is associated with a rule of the source code (i.e., the rule that the debugger identifies as buggy when it finds a buggy node in the ET). Taking into account that the final objective of the process is to find the program's rule which contains the bug—rather than a node in the ET—and considering that there is not a relation one-to-one between nodes and rules because several nodes can refer to the same rule, it is important to also consider the node's rules during the search. A first idea could be to explore first those subtrees with a higher number of associated rules (instead of exploring those subtrees with a higher number of nodes).

Example 3.3. Consider the following ET:



where each node is labeled with its associated rule and where the oracle answered NO to the question in the root of the tree. While heaviest first selects the right-most child because this subtree has four nodes instead of three, less YES first selects the left-most child because this subtree contains three different rules instead of two.

Clearly, this approach relies on the idea that all the rules have the same probability of containing the bug (rather than all the nodes). Another possibility could be to associate a different probability of containing the bug to each rule, e.g., depending on its structure: Is it recursive? Does it contain higher-order calls?.

The probability of a node to be buggy is $q \cdot p$ where q is the probability that the rule associated to this node is wrong, and p is the probability of this rule to execute incorrectly. Therefore, the probability P of a branch b to contain a bug is

$$P = 1 - \prod_{i=1}^{n} 1 - (q_i \cdot p_i)$$

where n is the number of nodes in b.

Clearly, if we assume that a wrong rule always produces a wrong result[10] (i.e., $\forall i \ . \ p_i = 1$), then the probability is

[10]This assumption is valid for instance in those flattened functional languages where all the conditions in the right-hand side of function definitions have been distributed between its rules. This is relatively frequent in internal languages of compilers, but not in source languages.

$$P = 1 - \prod_{i=1}^{r} 1 - q_i$$

where r is the number of rules in b.

Therefore, under the assumption that all the rules have the same probability of being wrong, given a bug in a program, the probability of being in branch b is $\frac{r}{R}$ where R is the number of rules in the program.

Hence, under the previous assumptions this strategy is (on average) better than heaviest first. For instance, in Example 3.3 the left-most branch has a probability of $\frac{3}{8}$ to contain a buggy node, while the right-most branch has a probability of $\frac{2}{8}$ despite it has more nodes.

However, in general, a wrong rule can produce a correct result, and thus it is important to also consider the probability of a wrong rule to return a wrong answer. This probability has been approximated by the debugger Hat Delta (see Section 3.10.8) by using previous answers of the oracle. The main idea is that a rule answered NO n times out of m is more likely to be wrong than a rule answered NO n' times out of m if $n' < n \leqslant m$.

Less YES First uses this idea in order to compute the probability of a branch to contain a buggy node. Hence, this strategy is a combination of the ideas from both heaviest first and Hat Delta. However, while heaviest first considers the structure of the tree and does not take into account previous answers of the user, Hat Delta does the opposite; thus, the advantage of less YES first over them is the use of more information (both the structure of the tree and previous answers of the user).

A direct generalization of Hat Delta for branches would result in counting the number of YES answers of a given branch; but this approach would not take into account the number of rules in the branch. In contrast, Less YES First proceeds as follows:

When a node is set correct, its associated rule and all the rules of its descendants are marked as correctly executed. If a rule has been executed correctly before, then it will likely execute correctly again. The debugger associates to each rule of the program the number of times it has been executed in correct computations based on previous answers. Then, when we have to select a child to ask, we can compute the total number of rules in the subtrees rooted at the children, and the total number of answers YES for every rule.

This strategy selects the child whose subtree is less likely to be correct (and thus more likely to be wrong). To compute this probability we calculate for every branch b a weight w_b with the following equation:

$$w_b = \sum_{i=1}^{n} \frac{1}{r_i^{(YES)}}$$

where n is the number of nodes in b and $r_i^{(YES)}$ is the number of answers YES for the rule r of the node i.

As with heaviest first, we select the branch with the biggest weight, the difference is that this equation to compute the weight takes into account previous answers of the user. Moreover, we assume that initially all the rules have been answered YES once, and thus, at the beginning, this strategy asks those branches with more nodes, but it becomes different as the number of questions asked increases.

As pointed out by Davie and Chitil [2006], independently of the considered strategy, taking into account the rule associated with each node can help to avoid many unnecessary questions (see the tree compression technique in Section 3.3).

With this strategy, the sequence of 9 questions asked for the ET in Figure 37 is:

```
 Starting Debugging Session...

(1)  main = False? NO
(2)  sqrtest [1,2] = False? NO
(4)  computs 3 = (9,9,8)? NO
(7)  comput2 3 = 9? YES
(16) comput3 3 = 8? NO
(20) partialsums 3 = [6,2]? NO
(21) sum1 3 = 6? YES
(23) sum2 3 = 2? NO
(24) decr 3 = 2? YES

Bug found in rule:
sum2 x = div (x + (decr x)) 2
```

3.10.11 *Divide by YES & Query* (Silva, 2006).

The same idea used in less YES first can be applied in order to improve divide & query. Instead of dividing the ET into two subtrees with a similar number of nodes, we can divide it into two subtrees with a similar weight. The problem that this strategy tries to address is the D&Q's assumption that all the nodes have the same probability of containing the bug. In contrast, this strategy tries to compute this probability.

By using the equation to compute the weight of a branch, this strategy computes the weight associated to the subtree rooted at each node. Then, the node which divides the tree into two subtrees with a more similar weight is selected. In particular, the node selected is the node which produces a least difference between:

—$w/2$ and the heaviest node whose weight is less than or equal to $w/2$
—$w/2$ and the lightest node whose weight is greater than or equal to $w/2$

where w is the weight of the suspicious area in the ET.

As with D&Q, different nodes could divide the ET into two subtrees with a similar weights; in this case, another strategy (e.g., Hirunkitti) can be used in order to select one of them.

We assume again that initially all the rules have been answered YES once. Therefore, at the beginning this strategy is similar to D&Q, but the differences appear as the number of answers increases.

Example 3.4. Consider again the ET in Example 3.3. Similarly to D&Q, the first node selected is the top-most "8" because only structural information is available. Let us assume that the answer is YES. Then, we mark all the nodes in this branch as correctly executed. Therefore, the next node selected is "2"; because, despite the subtrees rooted at “2” and “5” have the same number of nodes and rules, we now have more information which allows us to know that the subtree rooted at “5” is more likely to be correct since node “7” has been correctly executed before.

The main difference with respect to D&Q is that divide by YES & query not only takes into account the structure of the tree (i.e., the distribution of the program rules between its nodes), but also previous answers of the user.

With this strategy, the sequence of 5 questions asked for the ET in Figure 37 is:

```
Starting Debugging Session...

(7)  comput2 3 = 9? YES
(16) comput3 3 = 8? NO
(21) sum1 3 = 6? YES
(23) sum2 3 = 2? NO
(24) decr 3 = 2? YES

Bug found in rule: sum2 x = div (x + (decr x)) 2
```

3.10.12 *Dynamic Weighting Search* (Silva, 2006).

Subterm dependency tracking (see Section 3.10.9) relies on the idea that if a subterm is marked, then the error will likely be in the sequence of functions that produced and passed the incorrect subterm until the function where the programmer found it. However, the error could also be in any other equation previous to the origin of the dependency chain.

Silva [Silva and Chitil 2006] proposed a new strategy which is a generalization of subterm dependency tracking and which can integrate the knowledge acquired by other strategies in order to formulate the next question.

The main idea is that every node in the ET has an associated weight (representing the probability of being buggy). After every question, the debugger gets information that changes the weights and it asks for the node with a higher weight. When the associated weight of a node is 0, then this node leaves the suspicious area of the ET. Weights are modified based on the assumption that those nodes of the tree which produced or manipulated a wrong (sub)term, are more likely to be wrong than those that did not. Here, we compute weights instead of probabilities without loss of generality because this will ease a later combination of this strategy with previous ones. We assume initially that all the nodes have a weight 1 and that a weight 0 means “*out of the suspicious area*”.

Computing Weights from Subterms

Firstly, as with subterm dependency tracking, the oracle is allowed to mark a subterm from an equation as wrong (instead of the whole equation). Let us assume that the programmer is being asked about the correctness of the equation in a node n_1, and she marks a subterm s as wrong (or inadmissible). Then, the suspicious area is automatically divided into four sets. The first set contains the node, say n_2, that introduced s into the computation and all the nodes needed to execute the equation in node n_2. The second set contains the nodes that, during the computation, passed the wrong subterm from equation to equation until node n_1. The third set contains all the nodes which could have influenced the expression s in node n_2 from the beginning of the computation. Finally, the rest of the nodes form the fourth set. Since these nodes could not produce the wrong subterm (because they could not have influenced it), the nodes in the fourth set are extracted from the suspicious area and, thus, the new suspicious area is formed by the sets 1, 2 and 3.

Each subset can be assigned a different probability of containing the bug. Let us show it with an example.

Example 3.5. Consider the ET in Figure 37, where the oracle was asked about the correctness of equation 3 and it pointed out the computed subterm "`8`" as inadmissible. Then, the four sets are denoted in the figure by using different shapes and colors:

—**Set 1:** those nodes which evaluated the equation 20 to produce the wrong subterm are denoted by an inverted trapezium.

—**Set 2:** those nodes that passed the wrong subterm until the programmer detected it in the equation 3 are denoted by an ellipse.

—**Set 3:** those nodes that could influence the wrong subterm are denoted by a trapezium.

—**Set 4:** the rest of nodes are denoted by a grey rectangle.

The source of a wrong subterm is the equation which computed it. From our experience, all the nodes involved in the evaluation of this equation are more likely to contain the bug. However, it is also possible that the functions that passed this wrong term during the computation should have modified it and they did not. Therefore, they could also contain the bug. Finally, it is also possible (but indeed less likely) that the equation that computed the wrong subterm had a wrong argument and this was the reason why it produced a wrong subterm. In this case, this inadmissible argument should be further inspected. In the example, the wrong term "`8`" was computed because equation 20 had a wrong argument "`[6,2]`" which should be "`[6,3]`"; the nodes which computed this wrong argument have a trapezium shape.

Consequently, in the previous example, after the oracle marked "`8`" as wrong in equation 3, we could increase the weight of the nodes in the first subset with 3, the nodes in the second subset with 2, and the nodes in the third subset with 1. The nodes in the fourth subset can be extracted from the suspicious area because they could not influence the value of the wrong subterm and, consequently, their probability of containing the bug is zero (see Section 0**??**).

Given an ET, with a node containing an expression e marked as wrong, roughly, each of the four subsets is computed as follows:

—The equations in ET which belong to the Set 1 are those which are in the subtree rooted at the node that produced e (the origin of e).

—Given two nodes $n, n' \in$ ET where the expression e appears in n, e does not appear in the parent of n and node n' is the origin of e, then, the nodes in ET which belong to the Set 2 are all the nodes which are in the path between n and n' except n and n'.

—The Set 4 contains all the nodes that could not influence the expression e, and thus, it can be computed following the approach of [Silva and Chitil 2006]. In particular, Silva and Chitil introduce an algorithm which extracts a slice with respect to the expression e which contains all the nodes that could be the cause of e. Therefore, all the nodes that do not belong to the slice form Set 4.

—Finally, Set 3 contains all the equations in ET that do not belong to the Sets 1, 2 and 4.

Computing Weights From Rules

The information about the correctness of nodes which is provided by the oracle during the debugging session can be used to dynamically change weights.

As in Hat Delta, the main idea is that a rule answered NO n times out of m is more likely to be wrong than a rule answered NO n' times out of m if $n' < n \leqslant m$. Given a node n with associated rule r, its weight w is computed from the probability p of n to be wrong—based on the questions history—and the length of its 95% confidence interval:

$$w = p - (\frac{1.96}{\sqrt{n}}\sqrt{p\ (1-p)})^2$$

$$p = \frac{number\ of\ answers\ NO\ for\ r}{n = number\ of\ answers\ NO/YES\ for\ r}$$

For instance, if the debugger has to select a child A, B or C; where A is related to a rule that has been answered three times YES out of three questions, B is related to a rule that has been answered three times NO out of three questions, and C is related to a rule that has been answered three times YES and three times NO out of six questions; then B is more likely to be wrong than C (because $w_B > w_C$) and, in turn, C than A (because $w_C > w_A$). Hence, the debugger would ask first B and, with the new information, would recompute weights.

This strategy can integrate information used by other strategies in order to modify nodes' weights. In the next section we present an algorithm to combine information from different strategies.

3.10.13 *Comparing Algorithmic Debugging Strategies.*

A summary of the information used by all strategies is shown in Figure 40. The meaning of each column is the following:

—'*(Str)ucture*' is marked if the strategy takes into account the distribution of nodes (or rules) in the tree;

—'*(Rul)es*' is marked if the strategy considers the rules associated with nodes;

—'*(Sem)antics*' is marked if the strategy follows an order of semantically related questions, the more marks the more strong relation between questions;

—'*(Sub)expressions*' is marked if the strategy allows the user to mark subexpressions as wrong or inadmissible;

Strategy	Str.	Rul.	Sem.	Sub.	His.	Div.	Cost
Single Stepping	-	-	-	-	-	✓	n
Top-Down Search	-	-	✓	-	-	✓	$b \cdot d$
Top-Down Zooming	-	-	✓✓	-	-	✓	$b \cdot d$
Heaviest First	✓	-	✓	-	-	-	$b \cdot d$
Less YES First	✓	✓	✓	-	✓	-	$b \cdot d$
Divide & Query	✓	-	-	-	-	-	$b \cdot log_2 n$
Biased Weighting D&Q	✓	-	-	-	-	-	$b \cdot log_2 n$
Hirunkitti's D&Q	✓	-	-	-	-	-	$b \cdot log_2 n$
Divide by YES & Query	✓	✓	-	-	✓	-	$\underline{b \cdot d}$
Hat Delta	-	✓	-	-	✓	-	n
Subterm Dependency Track.	-	-	✓✓✓	✓	-	-	n
Dynamic Weighting Search	✓	✓	-	✓	✓	-	n

Fig. 40. Comparing algorithmic debugging strategies

—'*(His)tory*' is marked if the strategy considers previous answers in order to select the next node to ask (besides cutting the tree);

—'*(Div)isible*' is marked if the strategy can work with a subset of the whole ET. ETs can be huge and thus, it is desirable not to explore the whole tree after every question. Some strategies allow us to only load a part of the tree at a time, thus significatively speeding up the internal processing of the ET; and hence, being much more scalable than other strategies that need to explore the whole tree before every question. For instance, top-down can load the nodes whose depth is less than d, and ask d questions before loading another part of the tree. Note, however, that some of the non-marked strategies could work with a subset of the whole ET if they where restricted. For instance, heaviest first could be restricted by simply limiting the search for the heaviest branch to the loaded nodes of the ET. Other strategies need more simplifications: less YES first or Hat Delta could be restricted by only marking as correctly executed the first d levels of descendants of a node answered YES; and then restricting the search for the heaviest branch (respectively node) to the loaded nodes of the ET. Finally,

—'*Cost*' represents the worst case query complexity of the strategy. Here, n represents the number of nodes in the ET, d its maximum depth and b its branching factor.

If we analyze the information in Figure 40, we see that the costs of the first eight rows strictly depend on the structure of the ET (i.e., it can be determined by only analyzing the ET). In contrast, the cost of divide by YES & query, Hat Delta, subterm dependency tracking and dynamic weighting search also depends on the information provided by the user.

The cost of single stepping is too expensive. Its worst case query complexity is order n, and its average cost is $n/2$.

Top-down and its variants have a cost of $b \cdot d$ which is significantly lower than the one of single stepping. The improvement of top-down zooming over top-down is based on the time needed by the programmer to answer the questions; their query complexity is the same. Note that, after the tree compression described in

Section 3.3, no zoom is possible and, thus, both strategies are identical.

In contrast, while in the worst case the costs of top-down and heaviest first are equal, in the mean case heaviest first performs an improvement over top-down. In particular, on average, for each wrong node with b children $s_i, 1 \leq i \leq b$:

—Top-down asks $\frac{b+1}{2}$ of the children.

—Heaviest first asks $\frac{\sum_{i=1}^{b} weight(s_i) \cdot pos(s_i)}{\sum_{i=1}^{b} weight(s_i)}$ of the children.

where function *weight* computes the weight of a node and function *pos* computes the position of a node in a list containing it and all its brothers which is in descending order by their weights.

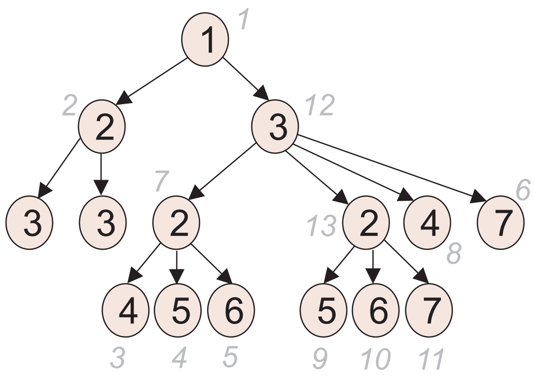
In the case of less YES first, the improvement is based on the fact that the heaviest branch is not always the branch with a higher probability of containing the buggy node. While heaviest first and less YES first have the same worst case query complexity, their average cost must be compared empirically.

D&Q is optimal in the worst case, with a cost order of $b \cdot (log_2\ n)$.

The cost of the rest of strategies is highly influenced by the answers of the user.

The worst case of Hat Delta happens when the branching factor is 1 and the buggy node is in the leaf of the ET. In this case the cost is n. However, in normal situations, when the ET is wide, the worst case is still close to n; and it occurs when the first branch explored is answered NO, and the information of NO answers obtained makes the algorithmic debugger explore the rest of the ET bottom up. It can be seen in Example 3.6.

Example 3.6. Consider the following ET:



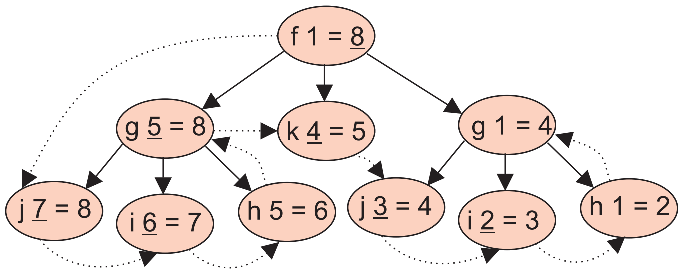
where each node is labeled with its associated rule. If we use the version of Hat Delta which counts correct computations, then when the oracle answers YES to a node, all its subnodes are marked as correctly executed once. This information could make the debugger to ask more questions than with top-down. In particular, a possible list of questions asked is specified by marking each node with its position in the list. This produces the following session: 1.- NO, 2.- YES, 3.- YES, 4.- YES, 5.- YES, 6.- YES, 7.- YES, 8.- YES, 9.- YES, 10.- YES, 11.- YES, 12.- NO, 13.- YES.

Despite subterm dependency tracking is a top-down version enriched with additional information provided by the oracle, this information—that we assume correct here to compute the costs—could make the algorithmic debugger to ask more questions than with the standard top-down. In fact, this strategy—and also dynamic weighting search if we assume that top-down is used by default—has a worst case query complexity of n because the expressions marked by the programmer can make the algorithmic debugger to explore the whole ET. The next example shows the path of nodes asked by the algorithmic debugger in the worst case.

Example 3.7. Consider the following program:

```
f x = g (k (g x))

g x = j (i (h x))

h x = x+1

i x = x+1

j x = x+1

k x = x+1
```

whose ET w.r.t. the call `f 1` is:



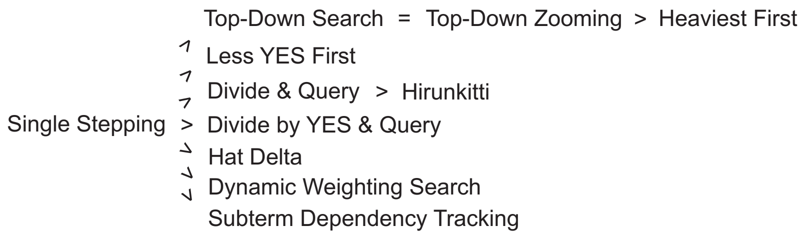
Here, the sequence of nodes asked with subterm dependency tracking is depicted with dotted arrows and the expressions marked by the programmer as wrong or inadmissible are underlined. Then, the debugging session is:

```
Starting Debugging Session...

f 1 = 8? NO  (The user selects "8")
j 7 = 8? YES (The user selects "7")
i 6 = 7? YES (The user selects "6")
h 5 = 6? YES
g 5 = 8? YES (The user selects "5")
k 4 = 5? YES (The user selects "4")
j 3 = 4? YES (The user selects "3")
```

```
i 2 = 3? YES (The user selects "2")
h 1 = 2? YES
g 1 = 4? NO

Bug found in rule:
g x = j (i (h x))
```

And, thus, all the ET's nodes are asked.

As a summary, the following diagram shows the relation between the costs of the strategies discussed so far:


Top-Down Search = Top-Down Zooming > Heaviest First
Less YES First
Divide & Query > Hirunkitti
Single Stepping > Divide by YES & Query
Hat Delta
Dynamic Weighting Search
Subterm Dependency Tracking

This diagram together with the costs associated to every strategy in Figure 40 allows algorithmic debuggers to automatically determine which strategy to use depending on the structure of the ET. This means that the algorithmic debugger can dynamically change the strategy used during a debugging session. For instance, after every question the new maximum depth (d) of the ET and its remaining number of nodes (n) is computed; if $d < log_2 n$ then heaviest first is used in order to select a new node, else Hirunkitti is applied. This result will be used in Section 0**??** to produce an algorithm which combines different strategies according to the dynamic changes in the structure of the ET's suspicious area during a debugging session.

4. A COMPARISON OF ALGORITHMIC DEBUGGERS

In this section, we do not focus on techniques, but in tools. The aim of this part of the survey is to overview current implementations of algorithmic debuggers and compare their functionality and performance.

4.1 Algorithmic Debuggers

The first part of our study consisted in the selection of the debuggers that were going to participate in the study. We found thirteen algorithmic debuggers, and we selected all of them except those immature enough to be used in practice (see Section 4.4). Our objective was not to compare algorithmic debugging-based techniques, but to compare mature and usable implementations. Therefore, we have evaluated each debugger according to its last implementation, not to its last report/article/thesis description.

The debuggers included in the study are the following:

Buddha:. Buddha[11] [Pope 2006] is an algorithmic debugger for Haskell 98 [Peyton-Jones 2003] programs.

[11]http://www.cs.mu.oz.au/∼bjpop/buddha

DDT:. DDT[12] [Caballero 2005] is part of the multiparadigm language TOY's distribution [López-Fraguas and Sánchez-Hernández 1999]. It uses a *graphical user interface* (GUI), implemented in Java (i.e., it needs the Java Runtime Environment to work).

Freja:. Freja[13] [Nilsson 1998] is a debugger for Haskell, result of Henrik Nilsson's thesis. It is able to debug a subset of Haskell; unfortunately, it is not maintained anymore.

Hat Delta:. Hat Delta[14] [Davie and Chitil 2006] belongs to Hat [Wallace et al. 2001], a set of debugging tools for Haskell. In particular, it is the successor of the algorithmic debugger Hat Detect.

B.i.O.:. B.i.O. (Believe in Oracles)[15] Curry [Hanus 2006] is a functional logic language. B.i.O. is a debugger integrated in the Curry compiler KICS [Braßel and Huch 2007], and can work as an algorithmic debugger.

Mercury's Algorithmic Debugger:. Mercury's compiler[16] [Somogyi et al. 1996] is a purely declarative logic and functional programming language. Its compiler has a debugger integrated with both a procedural debugger and an algorithmic debugger [MacLarty 2005].

Münster Curry Debugger:. One of the most important compilers for Curry is the Münster Curry compiler[17] [Lux 2006] which includes an algorithmic debugger.

Nude:. The NU-Prolog Debugging Environment (Nude)[18] [Naish et al. 1989] integrates a collection of debugging tools for NU-Prolog programs [Thom and Zobel 1988].

4.2 Functionality Comparison

In this section we study what of the functional requirements explained in Section 3 are currently implemented in the debuggers.

Table 4.2 presents a summary of the available functionalities of the studied algorithmic debuggers. Every column gathers the information of one algorithmic debugger. The meaning of the rows is the following:

—*Implementation Language*: Language used to implement the debugger.

—*Target Language*: Debugged language.

—*Strategies*: Algorithmic debugging strategies supported by the debugger: Top Down (TD), Divide & Query (DQ), Hat Delta's Heuristics (HD), Mercury's Divide & Query (MD), and Subterm Dependency Tracking (SD).

—*DataBase/Memoization*: Is a database used to store answers for future debugging sessions (inter-session memory)? Are questions remembered during the same session (intra-session memory)?

[12]http://toy.sourceforge.net
[13]http://www.ida.liu.se/∼henni
[14]http://www.haskell.org/hat
[15]http://www.informatik.uni-kiel.de/prog/mitarbeiter/bernd-brassel/projects/
[16]http://www.cs.mu.oz.au/research/mercury
[17]http://danae.uni-muenster.de/∼lux/curry/
[18]http://www.cs.mu.oz.au/∼lee/papers/nude/

—*Front-End*: Is it integrated into the compiler or is it standalone? Is it an interpreter/compiler or is it a program transformation?

—*Interface*: Interface used between the front-end and the back-end. If the front-end is a program transformation, then it specifies the data structure generated by the transformed program. Whenever the data structure is stored into the file system, brackets specify the format used. Here, DDT exports the ET in two formats: XML or a TXT. Hat Delta uses an ART (Augmented Redex Trail) with a native format. B.i.O. generates a list of step counts (see Section 3.9) which is stored in a plain text file.

—*Execution Tree*: When the back-end is executed, is the ET stored in the file system or in main memory? This row also specifies which debuggers produce the ET on demand.

—*Accepted Answers*: Yes (YE), No (NO), I Don't Know (DK), Inadmissible (IN), Maybe Yes (MY), Maybe Not (MN), and Trusted (TR).

—*Tracing Subexpressions*: Is it possible to specify that a (sub)expression is wrong?

—*ET Exploration*: Is it possible to explore the ET freely?

—*Tree Compression*: Does the debugger implement the tree compression technique?

—*Undo*: Is it possible to undo an answer?

—*Trusting*: Is it possible to trust modules (Mo), functions (Fu) and/or arguments (Ar)?

—*GUI*: Has the debugger a graphical user interface?

—*Version*: Evaluated version of the debugger.

4.3 Efficiency Comparison

We studied the growing rate of the internal data structure stored by the debuggers. This information is useful to know the scalability of each debugger and, in particular, their limitations with respect to the ET's size.

The study has proved that several debuggers are not usable with real programs because they run out of memory with long running computations. The main reason is that many of them store their ET in memory, and the size of the ET produces a memory overflow as soon as the computation is not medium-size. While the Münster Curry Debugger has the lowest growing rate in the size of its ET, the most scalable debugger is Hat Delta which successfully passed more benchmarks. The main advantages of the ART used by Hat Delta is that it is stored in secondary memory, and it shares data structures between different nodes.

In this experiment we have only considered those debuggers which already have a stable version and which are still maintained: Buddha, DDT, Hat Delta and Münster Curry debugger. We have omitted Mercury's debugger from this study because the collection of benchmarks used in the study are pure functional programs; and their translation to Mercury produced significant differences in the size and the structure of the ET which made it incomparable to the others.

In order to compare the growing rate of the ET's size of each debugger, we selected some benchmarks from the nofib-buggy suite [Silva 2007b] and we created other benchmarks which are particularly useful for algorithmic debugging because

\Debugger \ Feature\	**B.i.O.**	**DDT**	**Freja**	**Hat Delta**	**Buddha**	**Mercury Debugger**	**Münster Curry Debugger**	**Nue-Prolog**
Implementation Language	Curry Haskell	Toy (front-end) Java (back-end)	Haskell	Haskell	Haskell	Mercury	Haskell (front-end) Curry (back-end)	Prolog
Target Language	Curry	Toy	Haskell subset	Haskell	Haskell	Mercury	Curry	Prolog
Strategies	TD	TD DQ	TD	HD	TD	TD DQ SD MD	TD	TD
DataBase / Memoization	NO/NO	NO/YES	NO/NO	NO/YES	NO/YES	NO/YES	NO/NO	YES/YES
Front-end	Independent Prog. Tran.	Integrated Prog. Tran.	Integrated Compiler	Independent Prog. Tran.	Independent. Prog. Tran.	Independent Compiler	Integrated Compiler	Independent Compiler
Interface	Steps count (Plain text)	ET (XML/TXT)	ET	ART (Native)	ET	ET on Demand	ET	ET on Demand
Execution Tree	Main Memory on Demand	Main Memory	Main Memory	File System	Main Memory on Demand	Main Memory on Demand	Main Memory	Main Memory on Demand
Accepted answers?	YE NO DN	YE NO DN TR	YE NO MY MN	YE NO	YE NO DN IN TR	YE NO DN IN TR	YE NO	YE NO
Tracing subexpressions?	NO	NO	NO	NO	NO	YES	NO	NO
ET exploration?	YES	YES	YES	YES	YES	YES	NO	NO
Tree compression?	NO	NO	NO	YES	NO	NO	NO	NO
Undo	YES	NO	YES	NO	NO	YES	NO	NO
Trusting	Mo/Fu/Ar	Fu	Mo/Fu	Mo	Mo/Fu	Mo/Fu	Mo/Fu	Fu
GUI	NO	YES	NO	NO	NO	NO	NO	NO
Version	Kics 0.81893 (13.5.2008)	1.1 (2003)	1999-2000	2.05 (22.10.2006)	1.2.1 (01.12.2006)	Mercury 0.13.1 (01.12.2006)	0.9.10 (10.5.2006)	NU-Prolog 1.7.2 (13.07.2004)

they allow us to produce algorithmic debugging sessions with series of both huge and tiny questions. They also allow us to compare the debuggers with broad and long ETs. All these benchmarks together with the experiment results are publicly available at:

```
http://www.dsic.upv.es/~jsilva/algdeb
```

It is important to note that in this part of the study our objective was not to compare the debuggers against real programs. This is useless, because with the same ET, independently of its size, all the debuggers find the bug with the same number of questions (if they use the same strategy). Our objective was to study the behavior of the debuggers when handling different kinds of ETs. In particular, we produced deep, broad, balanced and unbalanced ETs with different sizes of nodes. With the experiment we were able to know how efficient the debuggers are when storing the ET in memory, and how scalable they are when the size of this ET grows up.

Once we collected the benchmarks for the experiment, we reprogrammed them for all the targeted languages of the debuggers. Finally, we tested the debuggers with a set of increasing input data in order to be able to graphically represent the growing rate of the size of the ET.

Since each debugger handles the ET in a different way, we had to use a different method in each case to measure their size:

—Hat Delta stores the ART into a file and traverses it during the debugging process. Then, we considered the size of the whole ART rather than the size of the implicit ET. Since the ART is used for other purposes than algorithmic debugging (e.g., tracing) it contains more information than needed.

—DDT saves ETs in two formats: TXT and XML. For our purpose, we used the size of the XML because this is the file loaded by the tool to make the debugging process.

—Münster Curry Debugger generates the ET in main memory. But we applied a patch—provided by Wolfgang Lux—that allows us to store the ET in a text file.

—Buddha also generates the whole ET in main memory. In this case, we used a shell script to measure the physical memory used by the data structures handled by the debugger. It might produce a slightly unfair advantage in that in-memory representation of the ET is likely to be more compact than any other representation stored on disk.

Figure 41 shows the results of one of the experiments. It shows the size of the ET produced by four debuggers when debugging the program *merge-sort*. X-axis represents the number of nodes in the ETs; Y-axis represents the ET's size in Kb; and Z-axis shows the debuggers. In this example we can see that Hat and Buddha supported big ETs without problems. However, DDT and the Münster Curry Debugger were not able to manage ETs with more than (approximately) two and ten thousand nodes respectively. DDT was out of memory and crashed. The Münster Curry Debugger needed a lot of time to generate big ETs, and we stopped it after 6 hours running.

It is important to note that we selected on purpose some benchmarks that process big data structures. The reason is that the size of the nodes is strongly dependent on the size of its data structures, and we wanted to study the growing rate with both small and big input data.

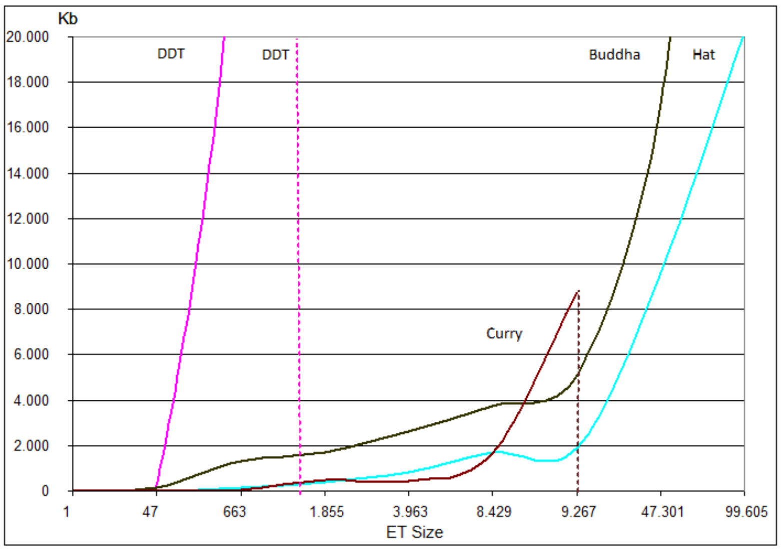


Fig. 41. Growing rate of declarative debuggers for merge-sort.

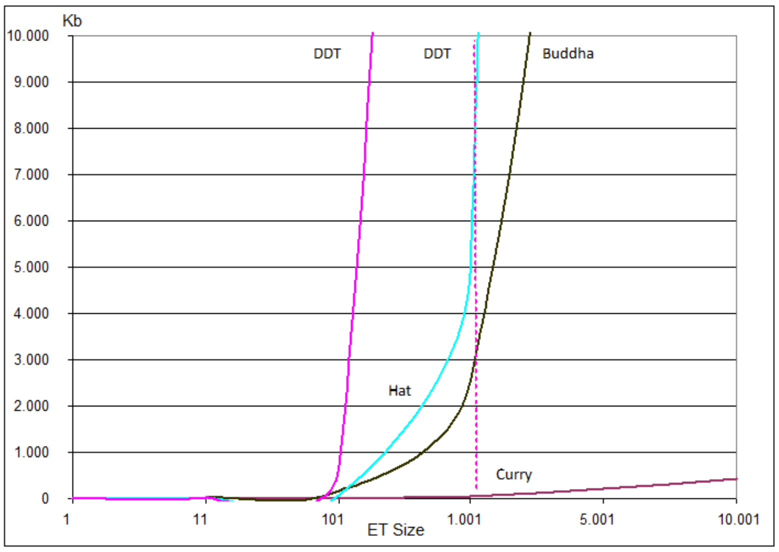


Fig. 42. Growing rate of declarative debuggers for factorial.

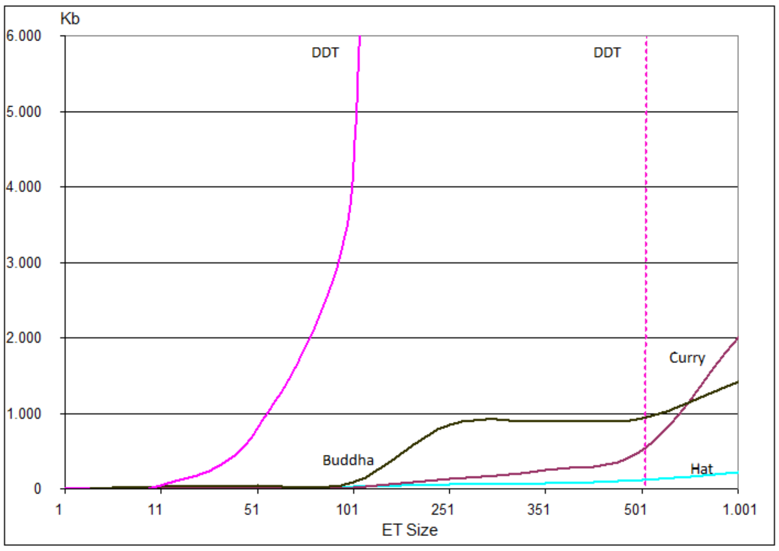


Fig. 43. Growing rate of declarative debuggers for length.

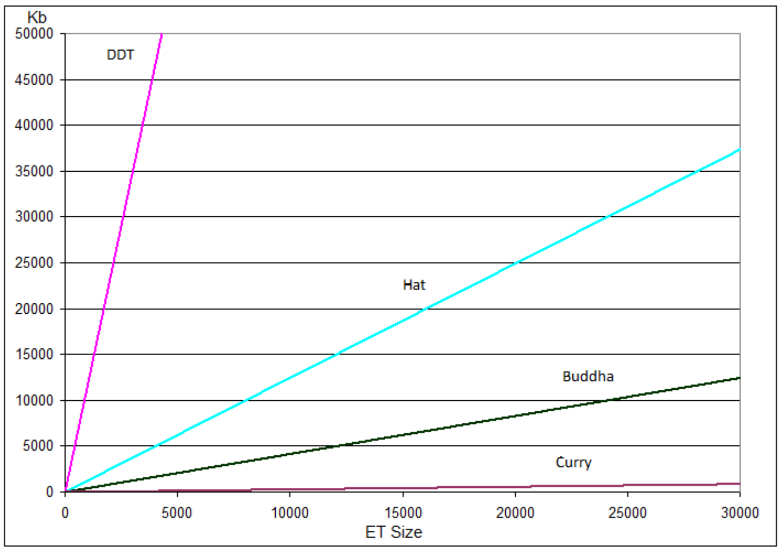


Fig. 44. Linear tendency graph of the growing rate of the ET's size.

Figures 42 and 43 show respectively the results obtained with the program *factorial* and *length*. They show two extreme cases: while, in *factorial* the impact of the input data over the size of the ET's nodes is not significant, in *length* it is very significant (we used lists up to one thousand elements).

Combining all the experiments, we computed the average linear tendency of the ET's size growing rate. It is depicted in Figure 44.

4.4 Other Debuggers

We did not include in our study those debuggers which do not have a stable version yet. This is the case, for instance, of the declarative debuggers for Java described in [Caballero 2006] and [Girgis and Jayaraman 2006]. We contacted with the developers of the Java Interactive Visualization Environment (JIVE) [Gestwicki and Jayaraman 2004] and the next release will integrate an algorithmic debugger called JavaDD [Girgis and Jayaraman 2006]. This tool uses the Java Platform Debugger Architecture to examine the events log of the execution and produce the ET.

We neither considered the debugger GADT [Fritzson et al. 1992] (which stands for Generalized Algorithmic Debugging and Testing). Despite it was quite promising because it included a testing and a program slicing phases, its implementation was abandoned in 1992.

The Prolog's debugger GIDTS (Graphical Interactive Diagnosis, Testing and Slicing System) [Kokai et al. 1999] integrated different debugging methods under a unique GUI. Unfortunately, it is not maintained anymore, and thus, it was discarded for the study.

Other debuggers that we discarded are the declarative debugger of the language Escher [Lloyd 1995], because it is not maintained since 1998; and the declarative debugger GraDE [Binks 1995] for the logic programming language Gödel which was abandoned.

We are aware of new versions of the studied debuggers which are currently under development. We did not include them in the study because they are not being distributed yet, and thus, they are not part of the current state of the practice. There are, however, two cases that we want to note:

(1) Hat Delta: The old algorithmic debugger of Hat was Hat Detect. Hat Delta has replaced Hat Detect because it includes new features such as tree compression and also improved strategies to explore the ET. Nevertheless, some of the old functionalities offered by Hat Detect have not been integrated in Hat Delta yet. Since these functionalities were already implemented by Hat Detect, surely, the next release of Hat Delta will include them. These functionalities are the following:
   - —Strategy Top-Down,
   - —undo capabilities,
   - —new accepted answers: Maybe Yes, Maybe No, and Trusted, and
   - —trusting of functions.

(2) DDT: There is a new β-version of DDT which includes many new features. These features are the following:
   - —New Strategies: Heaviest First, Hirunkitti's Divide & Query, Single Stepping, Hat Delta's Heuristics, More Rules First, and Divide by Rules & Query,
   - —tree compression, and
   - —a database for inter-session memoization.

5. CONCLUSIONS AND FUTURE WORK

The main conclusion of the study is that many techniques that have been studied and developed on a theoretical level, have not been implemented and/or integrated into usable algorithmic debuggers.

The functionality comparison has produced a precise description of what features are implemented by each debugger (hence, for each language). From the description, it is easy to see that many features which should be implemented by any algorithmic debugger are only included in one of them. For instance, only the Mercury debugger is able to trace subexpressions, only Hat Delta implements tree compression, only DDT has a GUI and only it allows the user to graphically explore the ET.

Another important conclusion is that none of the debuggers implement all the ET exploration strategies that appear in the literature. For instance, Hirunkitti's divide and query which is supposed to be the most efficient strategy has not been implemented yet.

Regarding the efficiency comparison, one conclusion is that the main problem of current algorithmic debuggers is not time but space. Their scalability is very low because they are out of memory with long running computations due to the huge growing rate of the ET. In this respect, the more scalable debugger of the four we compared are Hat Delta (i.e., it supported more debugging sessions than the others) and Buddha. However, the Münster Curry Debugger presents the lower ET's growing rate. It is surprising that much of the debuggers use a file to store the ET, and no debugger uses a database. The use of a database would probably solve the problem of the ET's size. Another technology that is not currently used and needs to be further investigated is the use of a clustering mechanism to allow the debugger exploring (and loading) only the part of the ET (the cluster of nodes) needed at each stage of the debugging session. Currently, no algorithmic debugger uses this technology that could both speed up the debugging session and increase the scalability of the debuggers.

Despite the big amount of problems we have identified, we can also conclude that there is a continuous progression in the development of algorithmic debuggers. This can be easily checked by comparing the dates of the releases studied and their implemented features. Also by comparing different versions of the same debuggers (for instance, from Hat Detect to Hat Delta many new functionalities have been added such as tree compression and new search strategies).

The comparison presented here only took into account objective criteria that can be validated. Subjective criteria that can be interpreted were omitted from the study. For instance, we did not talk about how easy to use or to install the debuggers are. This is another quirk of maturity, but we let this kind of comparison for future work.

This paper not only compares current implementations, but it also constitutes a guide to implement a usable algorithmic debugger, because it has established the main functionality and scalability requirements of a real program-oriented debugger.

ACKNOWLEDGMENTS

The author greatly thank Bernd Braßel, Sebastian Fisher, Wolfgang Lux, Lee Naish, Henrik Nilsson, and Bernie Pope for providing detailed and useful information about their debuggers. I also thank Olaf Chitil for planting the seed of this survey.

REFERENCES

Av-Ron, E. 1984. Top-down diagnosis of prolog programs. Ph.D. thesis, Weizmanm Institute.

Binks, D. 1995. Declarative Debugging in Gödel. Ph.D. thesis, University of Bristol.

Braßel, B. and Huch, F. 2007. The Kiel Curry system KiCS. In *Proc of 17th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2007) and 21st Workshop on (Constraint) Logic Programming (WLP 2007)*. Technical Report 434, University of Wrzburg, 215–223.

Caballero, R. 2005. A Declarative Debugger of Incorrect Answers for Constraint Functional-Logic Programs. In *Proc. of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming (WCFLP'05)*. ACM Press, New York, USA, 8–13.

Caballero, R. 2006. Algorithmic Debugging of Java Programs. In *Proc. of the 2006 Workshop on Functional Logic Programming (WFLP'06)*. Electronic Notes in Theoretical Computer Science, 63–76.

Conway, T., Henderson, F., and Somogyi, Z. 1995. Code Generation for Mercury. In *In Proc. of the International Logic Programming Symposium*. 242–256.

Davie, T. and Chitil, O. 2005. Hat-delta: One Right Does Make a Wrong. In *Draft Proceedings of the 17th International Workshop on Implementation and Application of Functional Languages (IFL'05)*, A. Butterfield, Ed. Tech. Report No: TCD-CS-2005-60, University of Dublin, Ireland, 11.

Davie, T. and Chitil, O. 2006. Hat-delta: One Right Does Make a Wrong. In *Seventh Symposium on Trends in Functional Programming, TFP 06*.

Fritzson, P., Shahmehri, N., Kamkar, M., and Gyimóthy, T. 1992. Generalized Algorithmic Debugging and Testing. *LOPLAS 1,* 4, 303–322.

Gestwicki, P. and Jayaraman, B. 2004. Jive: Java interactive visualization environment. In *OOPSLA '04: Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*. ACM Press, New York, NY, USA, 226–228.

Girgis, H. and Jayaraman, B. March 2006. JavaDD: a Declarative Debugger for Java. Tech. Rep. 2006-07, University at Buffalo.

Hanus, M. 2006. Curry: An Integrated Functional Logic Language (version 0.8.2 of march 28, 2006). Available at: `http://www.informatik.uni-kiel.de/∼curry/`.

Hirunkitti, V. and Hogger, C. J. 1993. A Generalised Query Minimisation for Program Debugging. In *Proc. of International Workshop of Automated and Algorithmic Debugging (AADEBUG'93)*. Springer LNCS 749, 153–170.

Kokai, G., Nilson, J., and Niss, C. 1999. GIDTS: A Graphical Programming Environment for Prolog. In *Workshop on Program Analysis For Software Tools and Engineering (PASTE'99)*. ACM Press, 95–104.

Lloyd, J. 1995. Declarative Programming in Escher. Tech. Rep. CSTR-95-013, Computer Science Department, University of Bristol.

López-Fraguas, F. and Sánchez-Hernández, J. 1999. TOY: A Multiparadigm Declarative System. In *Proc. of the 10th Int'l Conf. on Rewriting Techniques and Applications (RTA'99)*. Springer LNCS 1631, 244–247.

Lux, W. 2006. Mnster curry user's guide (release 0.9.10 of may 10, 2006). Available at: `http://danae.uni-muenster.de/∼lux/curry/user.pdf`.

MacLarty, I. 2005. Practical Declarative Debugging of Mercury Programs. Ph.D. thesis, Department of Computer Science and Software Engineering, The University of Melbourne.

Maeji, M. and Kanamori, T. 1987. Top-Down Zooming Diagnosis of Logic Programs. Tech. Rep. TR-290, ICOT, Japan.

NAISH, L. 1997a. A Declarative Debugging Scheme. *Journal of Functional and Logic Programming 1997,* 3.

NAISH, L. 1997b. A Three-Valued Declarative Debugging Scheme. In *Proc. of Workshop on Logic Programming Environments (LPE'97).* 1–12.

NAISH, L., DART, P. W., AND ZOBEL, J. June 1989. The NU-Prolog debugging environment. In *Proceedings of the Sixth International Conference on Logic Programming*, A. Porto, Ed. Lisboa, Portugal, 521–536.

NILSSON, H. 1998. Declarative Debugging for Lazy Functional Languages. Ph.D. thesis, Linköping, Sweden.

NILSSON, H. AND FRITZSON, P. 1994. Algorithmic Debugging for Lazy Functional Languages. *Journal of Functional Programming 4,* 3, 337–370.

PEREIRA, L. M. 1986. Rational Debugging in Logic Programming. In *Proc. on Third International Conference on Logic Programming.* Springer-Verlag LNCS 225, New York, USA, 203–210.

PEYTON-JONES, S. L., Ed. 2003. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press.

POPE, B. 2006. A declarative debugger for Haskell. Ph.D. thesis, The University of Melbourne, Australia.

SHAPIRO, E. 1982. *Algorithmic Program Debugging.* MIT Press.

SILVA, J. 2006. Algorithmic debugging strategies. In *Proc. of International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2006).* 134–140.

SILVA, J. 2007a. A Comparative Study of Algorithmic Debugging Strategies. In *Proc. of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'06).* Springer LNCS 4407, 143–159.

SILVA, J. 2007b. Nofib-buggy: The buggy benchmarks collection of haskell programs). Available at: `http://einstein.dsic.upv.es/darcs/nofib-buggy/`.

SILVA, J. AND CHITIL, O. 2006. Combining Algorithmic Debugging and Program Slicing. In *Proc. of 8th ACM-SIGPLAN International Symposium on Principles and Practice of Declarative Programming (PPDP'06).* ACM Press, 157–166.

SOMOGYI, Z., HENDERSON, F., AND CONWAY, T. 1996. The execution algorithm of mercury: an efficient purely declarative logic programming language. *Journal of Logic Programming 29,* 1-3, 17–64.

THOM, J. AND ZOBEL, J. 1988. Nu-prolog reference manual, version 1.3. Tech. Rep. 86/10, Department of Computer Science, University of Melbourne, Melbourne, Australia.

WALLACE, M., CHITIL, O., BREHM, T., AND RUNCIMAN, C. 2001. Multiple-View Tracing for Haskell: a New Hat. In *Proc. of the 2001 ACM SIGPLAN Haskell Workshop.* Universiteit Utrecht UU-CS-2001-23, 151–170.