Available online at www.sciencedirect.com
SCIENCE DIRECT
Information and Software Technology 48 (2006) 549–565
INFORMATION AND SOFTWARE TECHNOLOGY
www.elsevier.com/locate/infsof

Theory and algorithms for slicing unstructured programs

Mark Harman[a,*], Arun Lakhotia[b], David Binkley[c]

[a]Department of Computer Science, King's College, Strand, London WC2R 2LS, UK
[b]University of Louisiana at Lafayette, Lafayette, LA 70504, USA
[c]Loyola College, 4501 North Charles Street, Baltimore, MD 21210, USA

Received 21 January 2005; revised 31 May 2005; accepted 10 June 2005
Available online 30 August 2005

Abstract

Program slicing identifies parts of a program that potentially affect a chosen computation. It has many applications in software engineering, including maintenance, evolution and re-engineering of legacy systems. However, these systems typically contain programs with unstructured control-flow, produced using `goto` statements; thus, effective slicing of unstructured programs remains an important topic of study.

This paper shows that slicing unstructured programs inherently requires making trade-offs between three slice attributes: termination behaviour, size, and syntactic structure. It is shown how different applications of slicing require different tradeoffs. The three attributes are used as the basis of a three-dimensional theoretical framework, which classifies slicing algorithms for unstructured programs. The paper proves that for two combinations of these dimensions, no algorithm exists and presents algorithms for the remaining six combinations.

Keywords: Program slicing; Amorphous slicing; Unstructured control-flow

1. Introduction

Mark Weiser first defined a program slice in the context of debugging [35]. Since then program slicing has found many applications besides debugging [17,24,27,31] including program integration [23], comprehension [14,18] and reuse [3,12]. There also has been an active body of work in computing various types of slices resulting in a rich nomenclature for classifying slicing algorithms: intraprocedural vs. interprocedural; static vs. dynamic; backward vs. forward; executable vs. non-executable; and syntax-preserving vs. amorphous [6,7,13,21,33].

In short, intraprocedural slices consider a single procedure in isolation, while interprocedural slices consider multiple procedures with procedure calls. Static slices are computed from a program using static analysis while dynamic slices are computed from a program and an input and thus take into account a single execution of the program. A backward slice identifies program components that might affect a given computation. Its dual, a forward slice, identifies program components affected by a given component. An executable slice is an executable program that captures a subset of the original program's computation, while a non-executable (or closure) slice simply identifies the elements that affect (or are affected by) a given computation. These are often the same, but not always [4]. Finally, a syntax-preserving slice contains only portions of the original program's text, while an amorphous slice allows semantics-preserving transformations [18].

Slicing has found many applications because it allows the programmer to focus on a sub-computation; extracting it in the form of an executable subprogram—the slice. The sub-computation of interest may be one that the original author of the program had not considered and so the computation which denotes it may be arbitrarily scattered throughout the source code of the original program. The task of constructing the slice is thus the task of locating these scattered components and the supporting computations upon which they depend. It is a demanding problem because

* Corresponding author. Fax: +44 1895 251 686.
E-mail address: mark@dcs.kcl.ac.uk (M. Harman).

0950-5849/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2005.06.001

it requires a deep semantic analysis in order to ensure that the slice extracted preserves the behaviour of the original program with respect to the computation of interest.

The problem of slicing *unstructured* programs is important because many of the applications of slicing involve maintenance, evolution, and re-engineering of legacy systems, often written in programming styles which make heavy use of unstructured control flow [3,9,10,26]. Even recent systems contain a significant proportion of `goto` statements. For example, an inspection of Linux Kernel version 2.6.8.1 revealed that approximately 0.86% of the statements are `goto` statements. Finally, some programming languages (e.g. `C`), require the use of `break` statements in common constructions (such as the `switch` statement). The `break` statement denotes a limited form of unstructured control flow.

Ottenstein's Program Dependence Graph (PDG) based algorithm is currently the best known algorithm for intraprocedural slicing of structured programs [15,28]. This algorithm was not designed to compute slices of unstructured programs. Consequently, it fails to include any `goto` statements in a slice because a `goto` statement is neither the source of data nor control dependence. The literature contains several algorithms that extend Ottenstein's algorithm to compute slices of unstructured programs [1,2,11,20,25]. These algorithms are discussed in Section 6.

This paper focuses on the computation of static, backward, intraprocedural, executable slices of unstructured programs, henceforth simply referred to as slices. It makes the following contributions:

(1) Framework: The paper introduces the framework shown in Fig. 1 for classifying slicers of unstructured programs along three independent dimensions: termination behaviour, syntactic structure, and size. It is shown that slicing algorithms for two of the eight combinations within the framework, though desirable, simply do not exist. This non-existence result is not due to the familiar non-computability results relating to slice minimality [35]. Rather, it is a direct result of the particular properties of unstructured programs and their slices.
(2) Slicing algorithms: The paper presents slicing algorithms[1] for the remaining six combinations. These algorithms are built using common data structures, thereby facilitating examination of the tradeoffs present between the different possibilities. Finally, existing algorithms for slicing unstructured programs are placed into the framework. Interestingly, this reveals that, of the six possibilities, only three have been considered in previous slicing literature.

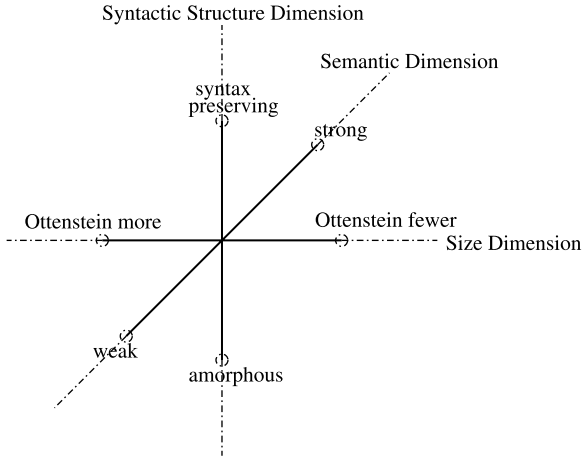

Fig. 1. The three dimensional framework.

The rest of the paper is organized as follows. Section 2 contains some necessary definitions. Section 3 presents the three-dimensions of the framework. Section 4 proves that two classes within the framework do not exist. Section 5 presents the new slicing algorithms. Section 6 discusses related work, places it into the framework, and compares it to the algorithms from Section 5. Finally, conclusions are presented in Section 7.

2. Definitions

This section defines the properties of the abstract syntax trees, control-flow graphs, and program dependence graphs required in subsequent sections. The language considered is essentially C, however the focus of the paper is on intraprocedural control issues; thus, the definitions and examples consider primarily assignment, `if-then-else`, `while`, `do-while`, sequence, `goto`, and 'special' statements. Interprocedural control issues (e.g. those introduced by `exit()`, `longjmp()`, or exceptions), and pointers, arrays, and other data dependence related features are mostly ignored.

2.1. Abstract Syntax Tree

The Abstract Syntax Tree (AST) is used to treat issues related to the order of statements in a program. This section defines the AST and two operators used to relate the ASTs of programs and their slices.

Definition 1. (Abstract Syntax Tree [16]). Each procedure P is represented by a standard Abstract Syntax Tree, denoted by AST(P). The relevant core of which is described as follows wherein bold text indicates the node kind (lower case) and contents (upper case), subordinate nodes appear

[1] The algorithms focus on the issue of unstructuredness and so issues to do with other language features (for instance pointer aliasing) are not considered.

between '[' and ']', '-list' is the postfix list operator, and ';' is the list construction operator.

Procedure
::= **procedure** [Statement-list]

Statement
::= **if C**
[Statement-list; "(implicit)goto endif"]
[Statement-list; label "endif"]
::= **while C** [label "top";
Statement-list;
"(implicit) goto top"]
::= **do-while C** [Statement-list]
::= **goto Label**
::= **skip**
::= **assignment Assignment-Expression**
::= **label Name**

Note that AST(P) differs from the standard definition in two respects. First, the condition of a branch node is maintained within the node, not as a child. Thus, an `if-then-else` statement has two children: a `then` child and an `else` child. All other control statements (e.g. `while` and `do-while` statements) have a single child: the body of the loop. Second, the AST includes *implicit* `goto` statements that are not part of the original source code. These represent two kinds of implicit control transfers: (a) the transfer from the end of the `then` part to the statement following an `if-then-else` statement, and (b) the transfer from the end of a loop body to the loop condition.

Example. The following code shows a program and a text representation of its AST including an implicit goto statement (appearing in italics).

Original program	Text representation of AST

if (x)
a=1
 	
else
b=2
	if (x)
	a=1
	goto endif
	else
	b=2
	endif:

This 'core' AST is used in all the formal parts of the paper. The results, however, apply to any larger language that contains this core AST.

Definition 2. (Syntax Order). *Syntax order* of AST(P), denoted by SO(P), orders the nodes of AST(P) in the order resulting from a left-to-right, top-to-bottom scan of P's source code.

Definition 3. (Projection of a Syntax Order). For a set of AST nodes N, the N *projection* of SO(P), denoted by SO(P)$|N$, is the sequence created by retaining (projecting out) from SO(P) only those nodes in N.

2.2. Control flow graph

The definition of a program's Control Flow Graph (CFG) given below is atypical in that it relaxes two reachability constraints. This definition is followed by two definitions based on the CFG. The first is used in the next section to formalize the size of a slice and the second is used to define the slicing algorithms in Section 5.

Definition 4. (Control Flow Graph [16]). The Control Flow Graph for program P, denoted by CFG(P), is a 4-tuple (N_P, E_P, n_{entry}, n_{exit}), where N_P is a set of nodes, $E_P \subseteq N_P \times N_P$ a set of edges, and n_{entry} and n_{exit} are unique start and end nodes, respectively, belonging to N_P. Herein, the standard definition is relaxed by removing the requirements that (1) a path exists from n_{entry} to every node in the CFG and (2) a path exists from every node to n_{exit}.

Definition 5. (Jumps). For a program P, $J_P \subseteq N_P$ denotes the set of `goto` (jump) nodes in CFG(P).

Definition 6. (Post Domination):

- CFG Node x *post dominates* node y iff all paths from y to n_{exit} contain x.
- CFG node x *immediately post dominates* node y iff all other post dominators of y also post dominate x.

2.3. Program dependence graph

Ottenstein and Ottenstein first noted that intraprocedural slices could be computed from the PDG using a simple graph traversal [28]. In essence, the statements that affect the computation represented by node n are those whose nodes are connected to n by paths of edges in the PDG. In the absence of `goto` statements, Reps and Yang demonstrate that projecting out the statements of the original program reached in this way, produces a syntactically correct program that captures a subset of the original program's semantics [30].

Definition 7. (Program Dependence Graph [28]). The PDG of a program P, denoted by PDG(P), contains essentially the same nodes as CFG(P) and edges that represent data dependences and control dependences (rather than the flow of control as in the CFG).

Definition 8. (Ottenstein Slice [28]). For program P and PDG node n, OttensteinSlice(P, n) = $\{x \mid$ there is a path in PDG(P) from x to $n\}$.

For program P and set of PDG nodes N, OttensteinSlice(P, N) = $\bigcup_{n \in N}$ OttensteinSlice(P, n).

Definition 9. (Executable Slice [30]). For program P and statement s, an executable slice of P taken with respect to s is a program obtained by projecting those statements of P whose nodes are in OttensteinSlice(P, PDG node for s).

3. The framework for slices and slicers

This section introduces the framework for classifying slices and slicing algorithms for unstructured programs. The framework has three orthogonal dimensions: termination behaviour, syntax, and size. In short these dimensions are described as follows:

Termination Behaviour. A slice is *strong* iff it terminates when the original program does, it is *weak* otherwise.

Syntax. A slice is *syntax preserving*, or *syntactic*, iff it is obtained solely by deleting statements from the original program. It is *amorphous* if it contains statements not in the original program.

Size. A slice is *Ottenstein-less*, *Ottenstein-equal*, or *Ottenstein-more* iff its size is less than, equal to, or more than the size of the slice produced by Ottenstein's PDG based slicing algorithm.

To simplify the sequel, slices are assumed to be taken with respect to the final value of a variable (represented in the PDG by a final-use vertex). These slices are referred to as being taken with respect to the variable. This treatment does not affect the results; it merely simplifies their presentation [30,8].

3.1. Termination

The termination dimension concerns the termination of P and its slices. Along this dimension a slice S may be either *strong* or *weak*. Informally, S is strong if it terminates for all inputs on which P terminates. It is weak if it may fail to terminate for some input on which P terminates. When P fails to terminate it is acceptable for S to also fail to terminate or for it to terminate. In the latter case the non-termination has, in essence, been 'sliced out.' The termination dimension is formalized in the following two definitions:

Definition 10. (Strong Equivalence). Program Q is *strongly equivalent* to program P with respect to a variable v iff for all initial states, whenever P terminates Q also terminates and both have the same final value for v.

Definition 11. (Weak Equivalence). Program Q is *weakly equivalent* to program P with respect to a variable v iff for all initial states, whenever P and Q both terminate they produce the same final value for v.

3.2. Syntactic

Along the framework's syntactic dimension, a slice may be *syntax-preserving* or amorphous. A syntax-preserving slice is created from the original program by simply deleting statements. Deletion of statements in a program does not change the syntactic order of the remaining statements. Thus, a syntax-preserving slice can be expressed using syntactic order as follows:

Definition 12. (Syntax Preserving). A program S is a *syntax-preserving* version of a program P iff $\mathrm{SO}(S) = \mathrm{SO}(P)|N_S$.

An amorphous slice may contain statements not in the original program. The use of the term 'amorphous' in the following definition deserves some clarification. Amorphous slicing was introduced by Harman and Danicic [19] and has since been developed by others [5,34]. A fully amorphous slice need bear no relation to the original program. In this 'fully amorphous' paradigm, the problem of slicing unstructured programs disappears; a program can simply be transformed into a `goto` free equivalent program, which can always be produced [29]. The transformed program can then be sliced using existing algorithms for structured programs. However, in this paper the degree of 'amorphous freedom' is highly restricted: an amorphous slice may only introduce `goto` statements not present in the original program.

Definition 13. (Amorphous). A program S is an *amorphous* version of a program P iff $\mathrm{SO}(S)|(N_S \backslash J_S) = \mathrm{SO}(P)|(N_S \backslash J_S)$.

3.3. Size

Along the size dimension, a slice may be *Ottenstein-less*, *Ottenstein-equal*, or *Ottenstein-more*. Ottenstein-less slices, though an interesting classification, are not considered further as their computation requires algorithms capable of producing smaller slices even for structured programs. Therefore, they are a separate issue. Of the two remaining sizes, a slice is Ottenstein-equal when it is the same size as an Ottenstein slice and Ottenstein-more when it is larger. As the Ottenstein algorithm never includes jump statements, they are factored out when comparing sizes. In the following definition, the size of X is denoted by $|X|$.

Definition 14. (Ottenstein-Equal). Let S be a slice of P taken with respect to variable v and $O = \mathrm{OttensteinSlice}(P, \mathrm{v})$. S is *Ottenstein-equal* if $|N_O| = |N_S \backslash J_S|$. (Recall that by construction J_O is empty.)

Definition 15. (Ottenstein-More). Let S be a slice of P taken with respect to variable v and $O = \mathrm{OttensteinSlice}(P, v)$. S is *Ottenstein-more* if $|N_O| < |N_S \backslash J_S|$.

3.4. The framework

The slicing classification framework has three dimensions based on the preceding six definitions. Using the following to denote these definitions (S) Strong (W) Weak (P) syntactically Preserving (A) syntactically Amorphous (E) Ottenstein-Equal, and (M) Ottenstein-More, slices and slicing algorithms are identified using the following lexicon to indicate the choice of each of the three dimensions.

$$\left\{\begin{matrix} S \\ W \end{matrix}\right\}\left\{\begin{matrix} P \\ A \end{matrix}\right\}\left\{\begin{matrix} E \\ M \end{matrix}\right\}\text{Slicing}$$

Note that since an algorithm that produces strong slices, by definition, produces weak slices, and an algorithm that produces syntax-preserving slices, by definition, produces amorphous ones, $SP\{E|M\}$ slices represent the most tightly constrained possibilities. In other words they are expected to be the most algorithmically challenging. In general, the smaller the slice the better for all applications of slicing. Therefore, of these two, it is clearly advantageous to have no additional nodes over-and-above those identified by the Ottenstein slice. This makes the most desirable choice SPE slicing. Unfortunately, as the next section shows, for unstructured programs SPE slicers do not exist. Furthermore, as will be shown, WPE slicers do not exist either.

3.5. Applications of the different dimensions

As this paper shows, there is a trade-off between possible properties of a program slice. These choices are reflected in the algorithms that the paper introduces for producing forms of slice which favour one property over another. The preferred choice of slicing algorithm depends on the target application. In this section, three possible choices of slicing are considered and their possible applications described.

As the example applications in these three sub-sections illustrate, there is a choice to be made when considering the precise form of slicing to apply. The choice involves a trade-off as it is impossible to have a slice which is always syntax-preserving, termination preserving and has no more nodes than the Ottenstein slice. However, in each case, there is a case to be made concerning which property can be conceded and why the other two are more important.

3.5.1. Applications of WAE slicing

For debugging applications [24,27], Weak, Amorphous, Ottenstein-Equal (WAE) slices are likely to be the most acceptable. This is because the use of slicing in debugging is geared towards reducing the number of nodes which have to be considered by the (human) debugger in order to identify the location and cause of the fault.

In debugging, the original program is executed. When an error is observed the first task is to identify the part of the program which may have caused the error to manifest itself. Slicing can help in this activity because it reduces the size of the program, allowing the debugging activity to focus upon those lines which may have caused the error. Therefore, for this application, it will be most important to limit the size of the slice, but less important to be able to execute it (the original program having already been executed and having produced an error).

Since the slice will not be executed, but merely inspected by a human, it is not important that the slice preserve termination (so it may be weak), nor will it matter if the slice introduces some `goto` nodes to direct control flow (so it may be amorphous). It will matter more, should the slice add too many additional nodes. Therefore, WAE slicing would seem most suitable, because the strong version (the Strong, Amorphous Ottenstein-Equal slice) will generally require more additional `goto` nodes in order to guarantee that it is Strong.

3.5.2. Applications of SAE slicing

In re- and reverse- engineering applications [3,10,26], Strong, Amorphous, Ottenstein-Equal (SAE) slices would appear to be the most suitable form of slicing. This is because reverse engineering involves extraction of reusable components to be re-integrated into the reverse engineered system.

Clearly, as the slice will become part of a system to be executed, it must preserve termination (must be strong) and (as with all slicing applications) it would be desirable to have as few additional nodes as possible (must be Ottenstein Equal). However, since re-engineering a system involves changing its syntax, there would appear to be no reason to require that the slice be completely faithful to the syntax of the original, so it may be amorphous.

In reverse engineering, the trade off is size for amorphousness. However, it is not possible to concede termination behaviour, so in order to ensure that the slice is as small as possible, it may be necessary to concede syntax preservation. That is, the slice will be as small as possible (while preserving termination characteristics), but may not preserve syntax. Since reverse engineering involves changing the syntax of the program in any case, this concession to amorphousness (lack of syntax preservation) is acceptable.

3.5.3. Applications of SPM slicing

For program integration [8,23], Strong, syntax-Preserving, Ottenstein-More (SPM) slices are most likely to be suitable. This is because the integration of two versions of a program must be understood by the original programmers.

However, it will be also executed, so it must preserve termination (so must be strong). As the framework shows, this leaves no alternative but to allow the slice to be Ottenstein-More, even though this is undesirable. Though undesirable, it is the least of three evils. As this paper shows, it is sometimes necessary for a Strong, syntax-Preserving slice to be Ottenstein-More.

4. Non-existence of SPE and WPE slicers

This section demonstrates that two classifications within the framework simply do not exist. These two are WPE (Weak, syntax-Preserving, Ottenstein-Equal) slicers and SPE (Strong, syntax-Preserving, Ottenstein-Equal) slicers. This section also shows that there exist programs and slicing criteria for which WPE slices exist, but SPE slices do not.

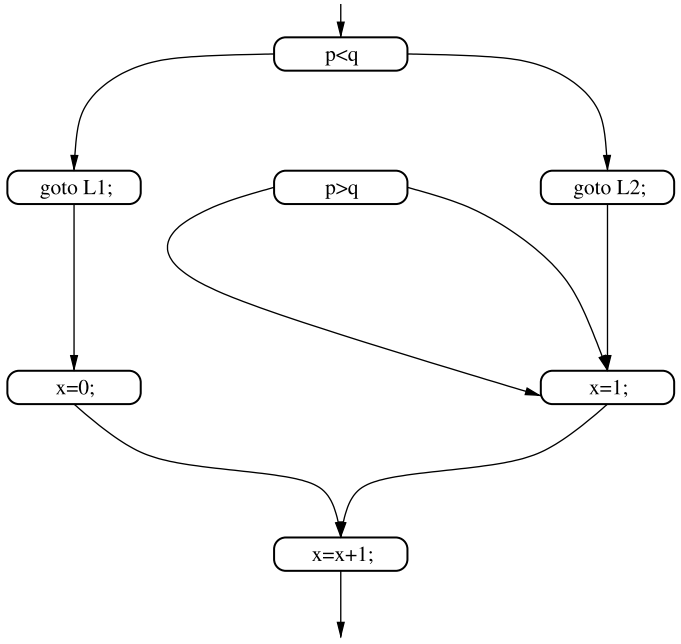

Fig. 2. A program and its CFG for which WPE slices do not exist.

These results show that it is necessary to choose between these three desirable properties: one simply cannot have a slice which is both syntax and termination preserving and which contains no more nodes than the Ottenstein slice.

Proposition 1. *WPE slicers do not exist*.

Proof 1. Consider the program fragment and its CFG shown in Fig. 2. The Ottenstein slice taken with respect to the final value of `x` is {1, 6, 7, 8} (shown in bold in the figure). Clearly the `goto` nodes at Lines 2 and 3 need to be included in order for the slice to have the correct semantics. However, their inclusion alone makes the resulting set of nodes neither a WPE slice nor an SPE slice. The problem occurs because of the transfer of control from Node 6. In the original program, control transfers from Node 6 to Node 8. Without the inclusion of Node 4, the transfer of control (an 'implicit goto') will not occur. Node 4 does not control anything other than a goto node, and therefore, no criterion which will lead to its inclusion using the Ottenstein Algorithm. In such a situation, there are two choices:

- Include the predicate node that directly contains the implicit `goto`.
  This will produce a slice that is not Ottenstein-equal. (In this case, Nodes 2, 3 and 4 would be added to the Ottenstein slice, making it an SPM slice.)
- Add a new `goto` to make the implicit transfer of control explicit.
  This will produce a slice which is not syntax preserving. (In this case, the new node 'goto end' would be added immediately after Node 6, making the slice SAE.) □

Given that Line 4 of Fig. 2 is a predicate which controls only goto nodes, one might conclude that this result is only of theoretical interest because it holds only when unrealistic branch nodes are involved. This is in fact not the case, nor is it the case with other examples in this section. For example, the following code, from the Linux Kernel open file routine, includes such a `goto` node. The conditional '`if (IS_ERR(f))`' controls only '`goto out_error;`'.

```
if (!IS_ERR(tmp)){
  lock_kernel();
  fd =get_unused_fd();
  if (fd >=0) {
    struct file ( f=filp_open(tmp,
    flags, mode);
    error =PTR_ERR(f);
    if (IS_ERR(f))
      goto out_error;
    fd_install(fd, f);
  }
  out:
    unlock_kernel();
    putname(tmp);
  }
  return fd;
  out_error:
    put_unused_fd(fd);
    fd =error;
    goto out;
```

Corollary. *SPE slicers do not exist*.

Proof 2. Since the set of strongly equivalent programs is contained within the set of weakly equivalent programs, the non-existence of WPE slicers implies the non-existence of SPE slicers. In Fig. 2, all syntactic subsets of the program

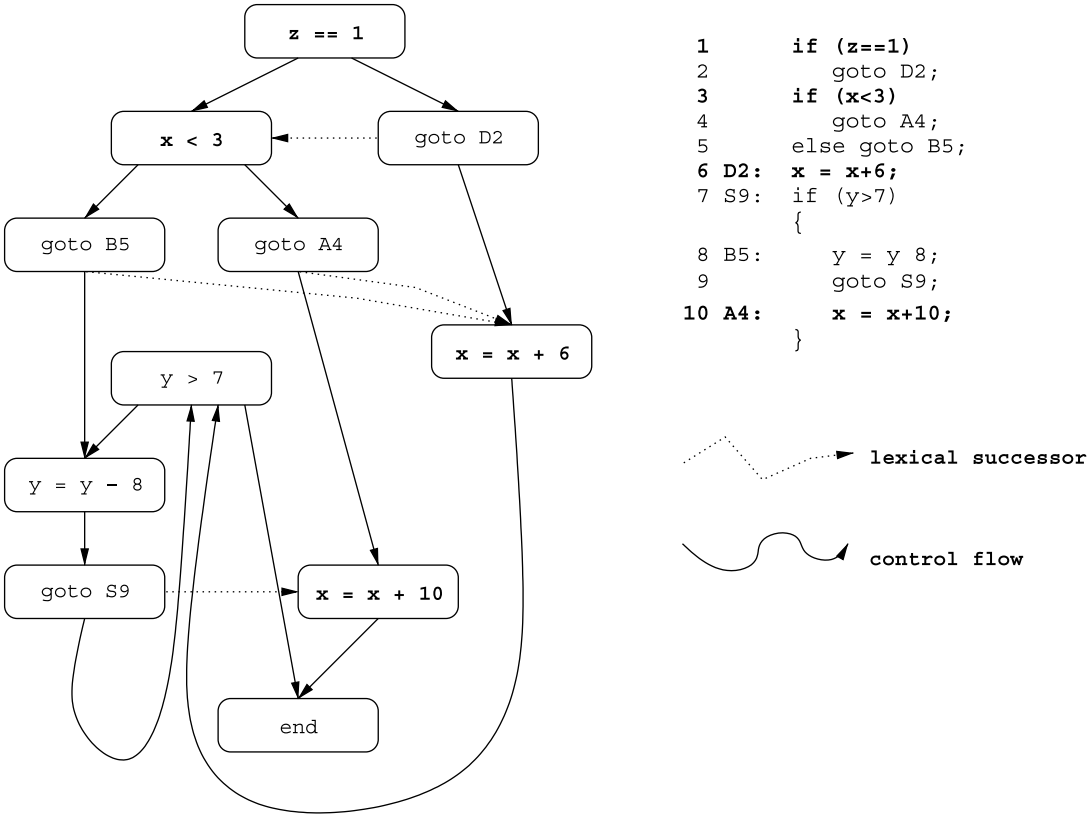

Fig. 3. A program with CFG for which no SPE slice exists, but for which WPE slices does exist.

terminate, so all weak slices are, by definition, also strong slices. □

This section closes by considering the question of whether there are programs for which SPE slices (not slicers) do not exist, but for which WPE slices do exist. The answer is 'yes', as the following theorem demonstrates.

Proposition 2. *There exist programs for which WPE slices exist, but SPE slices do not.*

Proof 3. Consider the program shown in Fig. 3. The Ottenstein slice of this program at the final use of `x` is {1, 3, 6, 10}. Suppose an attempt is made to construct a slice using only the nodes of the Ottenstein slice and the goto nodes from the original program. The crucial node is Node 9. The various options are

- Include Node 9, the branch controlling the loop, and all its transitive dependences. This will include Nodes 7 and 9 in the slice, making it SPM.
- Replace the `goto` at Node 9 by a new `goto` node, `goto end`. This makes the slice SAE.
- Include Node 9 in the slice. The resulting slice will fail to terminate, where the original program would have terminated, making it WPE.
- Do not include Node 9 in the slice. The resulting program is not a slice because from Node 9 execution will pass (incorrectly) from Node 6 to Node 10.

For this program, any slice taken with respect to the final use of x must either be weak, Ottenstein-more, or amorphous. In particular, note there is a WPE slice and yet no SPE slice. □

5. Slicing algorithms

As shown in the previous section, a slicer has to produce either amorphous or Ottenstein-more slices because Proposition 1 demonstrates that WPE and SPE slicers do not exist. Furthermore, the existence of WAE and SAE algorithms, presented in this section, obviates the need to consider WAM and SAM algorithm. The latter can be produced by simply adding the Ottenstein slice taken with respect to any node not in the WAE or SAE slice, respectively, which has the effect of unnecessarily increasing slice size without affecting the semantics of the original slice. These two observations reduce the framework's eight possible classifications for slicing algorithms to the four interesting classifications considered in this section.

Each of the four algorithms is presented in a declarative style. This makes it easier to understand and clarifies the two key choices: determining how to handle termination and implicit `goto`s. Each algorithm consists of two stages. The first stage, described in Section 5.2, produces a projection 4-tuple. The second stage, described in Section 5.1, uses this

4-tuple to create the AST of the resulting slice. The algorithm for the second stage is described first, since it motivates the construction of the 4-tuple. A projection 4-tuple identifies four sets of statements and is defined as follows:

Definition 16. (Projection 4-tuple). A projection 4-tuple (*S, G, M, X*), denoted by ψ, contains

- *S*, the nodes of an Ottenstein slice (recall that an Ottenstein slice never includes a `goto` node).
- *G*, a set of `goto` nodes to be preserved in the slice.
- *M*, a set of branch nodes to be morphed (transformed into `goto` statements by Phase 2).
- *X*, a set of branch nodes to be retained in the slice, but which are not members of the Ottenstein slice.

The respective elements of a 4-tuple are denoted by $\psi.S$, $\psi.G$, $\psi.M$, and $\psi.X$. Note that each pairwise intersection of these four sets is empty.

5.1. AST Projection

The second stage, the AST projection algorithm, is shown in Fig. 4. Informally, the ψ-projection of AST(*P*), denoted by AST(*P*)$|\ \psi$, is the AST resulting from performing the following:

- replace with a skip node all nodes not in $\psi.S \bigcup \psi.G \bigcup \psi.M \bigcup \psi.X$, and
- replace $m \in \psi.M$ with the node goto *ipd*, where *ipd* is the immediate postdominator of *m*.

The AST projection algorithm does not remove nodes from the tree. It merely replaces the nodes to be deleted with `skip` statements, which is semantically equivalent to deleting the statement. Replacing a branch statement by skip does not involve the deletion of its children. Instead, they are replaced by a subtree in which implicit `goto`'s

AST$(P)|\ \psi =$
Let $(S, G, M, X) = \psi$.
For each node n in AST(P) do
(1) if $n \notin S \cup G \cup X$ and is not a branch statement then
replace n with *skip*.
(2) if $n \notin S \cup M \cup X$ and is a branch statement then
(a) if n is an `if-then-else` statement then
replace the subtree rooted at n with the subtree representing the sequence `skip` ; T ; E,
where T is the `then` child of n and E is the `else` child of n.
(b) if n is an `if-then` or a `while` statement then
replace the subtree rooted at with the subtree representing the sequence `skip` ; B,
where B is the child of n.
(c) if n is a `do-while` statement then
replace the subtree rooted at with the subtree representing the sequence B; `skip` ,
where B is the child of n.
(3) if $n \in M$ and is a branch statement then
Let L' be the immediate postdominator of n in CFG(P)
(a) if n is an `if-then-else` statement then
replace the subtree rooted at n with the subtree representing the sequence `goto` L' ; T ; E,
where T is the `then` child of n and E is the `else` child of n.
(b) if n is an `if-then` or a `while` statement then
replace the subtree rooted at n with the subtree representing the sequence `goto` L' ; B,
where B is the child of n.
(c) if n is a `do-while` statement then
replace the subtree rooted at n with the subtree representing the sequence B; `goto` L' ,
where B is the child of n.
Return the new AST.

Fig. 4. An Algorithm for AST Projection.

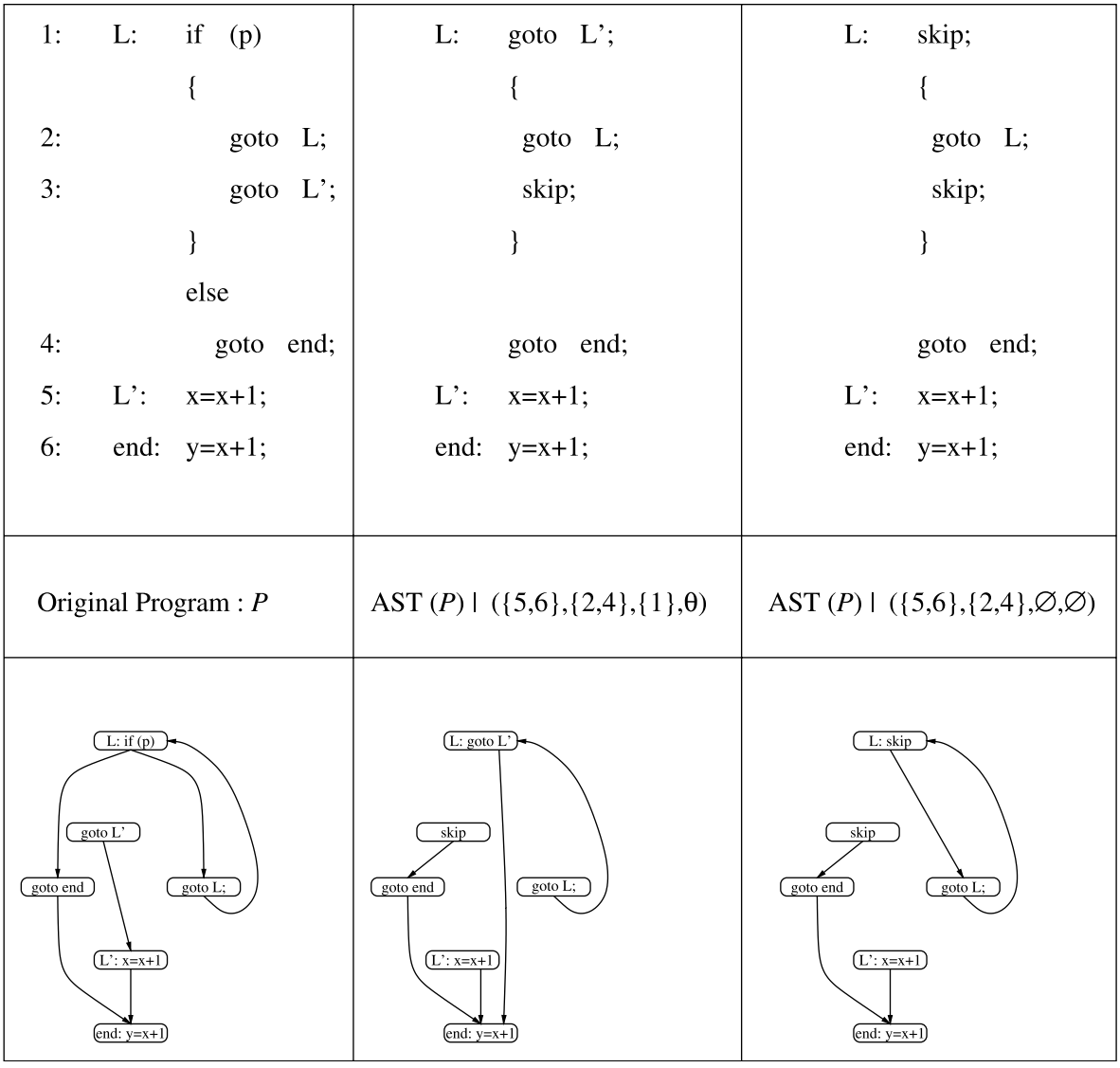

Fig. 5. Examples of AST Projection.

have been made explicit. For example, an `if-then-else` node is replaced by a sequence that composes its `then` subtree and its `else` subtree separated by a, now-explicit, `goto` statement. The newly explicit `goto` statement prevents an inadvertent transfer of control from the end of the `then` subtree to the start of the `else` subtree.

The following three examples help to illustrate the projection algorithm. The first two are based on the program fragment shown in the top left section of Fig. 5. The program has one branch (Line 1) and one implicit `goto` (Line 3). Consider its projection with $\psi = (S, G, M, X) = (\{5, 6\}, \{2, 4\}, \{1\}, \varnothing)$. The AST produced by projecting this program fragment with respect to ψ is depicted in the center column. In this projection, the branch (Line 1) is in the 'morph branch' set $\psi.M$, so AST projection 'rewires' the control flow of the program so that control passes from Line 1 to its immediate post dominator (Line 5) via the freshly introduced `goto` statement '`goto L`'. The only other modification performed by the projection is the replacement of the implicit `goto` (Line 3), which is not in $\psi.G$, with `skip`. The resulting program is a SAE slice.

As a second example, consider the AST projection of the same program with $\psi = (\{5, 6\}, \{2, 4\}, \varnothing, \varnothing)$. The AST produced by projecting the original code fragment with respect to ψ is depicted in the rightmost column of the figure. In this projection, the 'morph branch' set $\psi.M$ is empty, which leads to Line 1 being replaced with `skip`. Replacing this branch by `skip` creates a non-terminating cycle consisting of Lines 1 and 2; thus the resulting program is a weak slice. As with the first example, the implicit `goto` (Line 3) is not included in $\psi.G$ and so it is replaced by skip in the projected AST. This is an example of WAE slice.

As an example of a WPM slice (and also a SPM slice), consider the program in Fig. 6. Its AST Projection for $\psi = (\{1, 3, 10, 11, 12\}, \{2, 4, 6, 8\}, \varnothing, \{5\})$ mostly follows the same pattern as the above examples. However, with

```
1:    S1:  if (c)
           {
2:              goto S5;
           }
3:    S2:  if (p<q)
           {
4:              goto S4;
           }

5:    S3:  if (p>q)
           {
6:              goto S7;
7:    S4:       x = 0;
8:              goto S6;      // implicit goto node, shown explicitly
           }
           else
           {
9:              goto S7;
10:   S5:       x = 1;
           }
11:   S6:  x = x + 1;
12:   S7:  x = x + 2;
```

Fig. 6. An example for WPM slice.

$\psi.X = \{5\}$, Line 5 is kept in the projected program even though it is not in the Ottenstein slice. The resulting projected slice include Lines {1, 2, 3, 4, 5, 6, 8, 10, 11, 12}. Because Ottenstein slice of Line 5 is not in $\psi.S$, if there were any statement that controlled Line 5 those statements may not be in the projection. Hence, if control reaches Line 5 the program's state may not be consistent with what is expected at Line 5. That makes the slice weak, in the general case, because the program may terminate abnormally or not terminate at all.

5.2. Building ψ

The algorithms for building ψ must account for certain liveness properties. Correctly accounting for these allows the slicing algorithms to avoid including unnecessary program components. The following three definitions are used to capture the required liveness conditions.

Definition 17. (Live Edges). Given a set of CFG nodes S, an edge from a branch node b to its non-syntactic successor (e.g. the first statement in the else block of an if-then-else) is live only if $b \in S$. All other CFG edges are always live.

Definition 18. (Inward Goto [29]). A goto statement is said to be *inward* if it jumps into a control structure.

For example, the statement '`goto out`' at the end of the Linux code shown in Section 4 is an inward `goto`.

Definition 19. (Live `goto` Nodes). Given a program P and a collection of non-`goto` nodes S from CFG(P), the set of *live* `goto` *nodes*, denoted by LiveGotos(P, S), is the union of the following two sets:

(1) those *explicit* `goto` nodes reachable from a node in S by a path composed only of edges *live* with respect to S.
(2) those *implicit* `goto` nodes g reachable from a node in S by a path composed of edges *live* with respect to S such that this path contains an inward `goto` g' that is not (transitively) nested within the same branch of the conditional or the loop directly enclosing g.

The program in Fig. 7 gives examples of the two kinds of live goto nodes. The goto statements at Lines 7 and 12 are implicit gotos, explicitly represented in the program for exposition purposes. Of these, Line 12 is live with respect to Ottenstein slice, but Line 7 is not. Line 12 is live because (a) the path (1, 3, 4, 11, 12) consists of live edges (b) this path contains Line 4, an inward `goto`, and (c) Line 12 is nested in the true branch of Line 9, but Line 4 is not (transitively) nested in this branch. Line 7 is not live because no path leading to it contains an inward `goto`.

5.2.1. Weak, amorphous, Ottenstein-equal (WAE) slices

The first of the four slicing algorithms computes WAE slices. A WAE slice is created by adding certain goto nodes to an Ottenstein slice. Simply adding all the goto nodes would suffice, but would be a somewhat facile course of action. Instead, the set of *live gotos* captures a sufficient subset of the `goto` nodes. The first stage of a WAE slice is computed by Function WAESlice(P, V) (see Fig. 8), which yields the 4-tuple ψ described by the following equations:

$\psi.S = \text{OttensteinSlice}(P, V)$
$\psi.G = \text{LiveGotos}(P, \psi.S)$
$\psi.M = \emptyset$
$\psi.X = \emptyset$

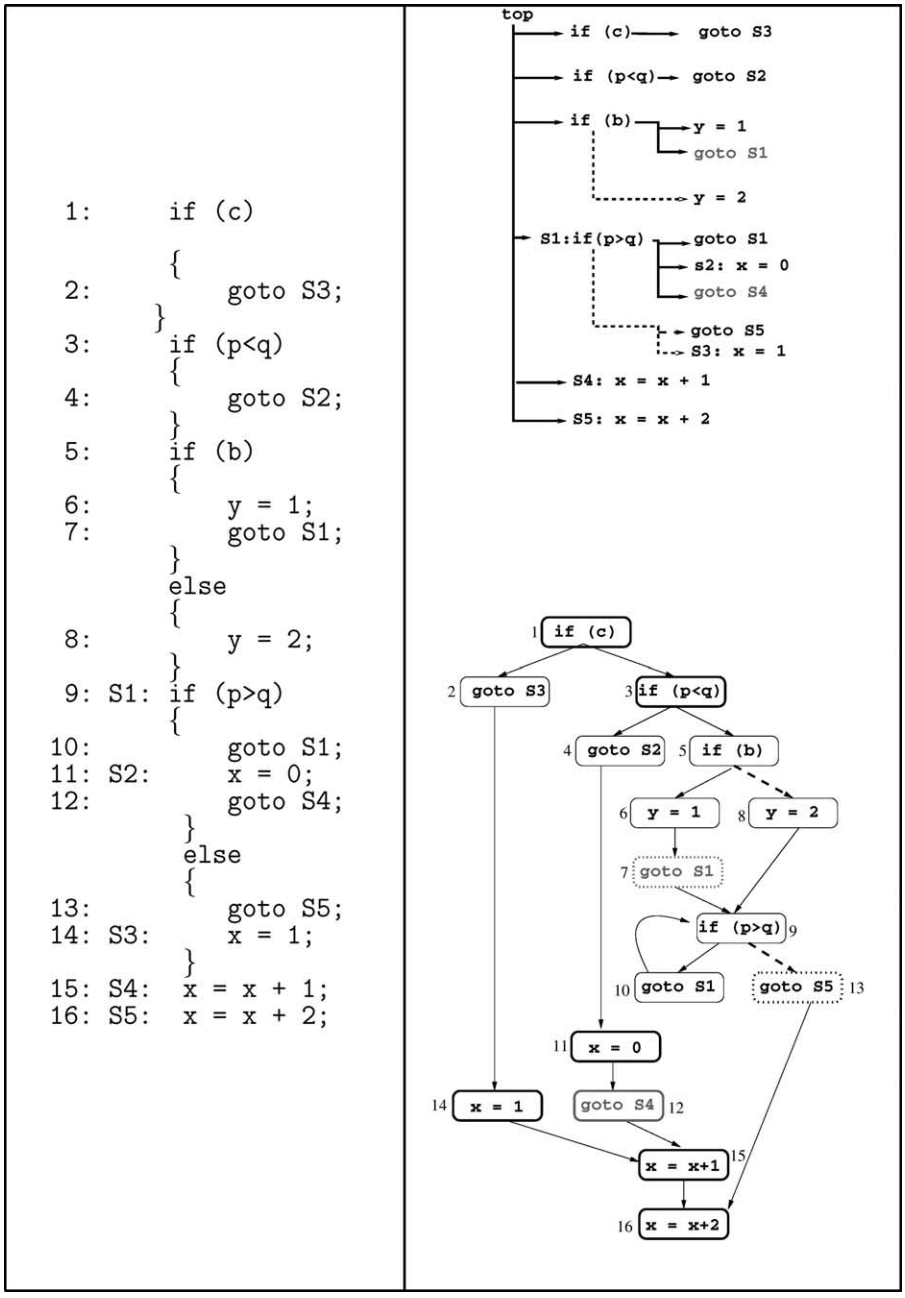

Fig. 7. A program, its AST, and its CFG used to illustrate explicit and implicit live gotos. In the AST and CFG dashed edges are the false branches of conditionals and implicit gotos (Lines 7 and 12) are shown in gray. In the CFG, bold nodes are in the Ottenstein Slice taken with respect to x at Line 16 and nodes for gotos that are not live are shown dotted.

For example, in the slice illustrated in Fig. 7, the explicit `goto` statement at Line 13 is not live because the `false` branch in which it is contained is not live and this `goto` statement cannot be reached without traversing that branch. As a consequence, the goto statement at Line 13 (like the non-live goto at Line 7) will not be retained in the WAE slice. The resulting slice is weak because the `goto` statement in Line 10 creates a loop after the `if` statement at Line 9 is replaced by skip.

The semantic guarantee of an Ottenstein slice [30] holds for those statements represented in $\psi.S$ provided that the slice preserves projections of paths from the original program's CFG. The set $\psi.G$ contains the necessary `goto` nodes to ensure the preservation of these paths and thus the semantic correctness of the statements represented in $\psi.S$. The slice is weak because paths that traverse the `false` edge of an `if` statement are not preserved in the slice and their absence leads to non-termination. For example, the right column of Fig. 5 demonstrates this. The slice is amorphous as the set $\psi.G$ may contain implicit `goto` nodes whose corresponding branch node is not in $\psi.S$. These implicit `goto` nodes must be made explicit in the slice; thus, introducing statements that

```
WAE_Slice(P, V) =
    local S, G
    S := OttensteinSlice(P, V)
    G := LiveGotos(P, S)
    return (S, G, ∅, ∅)
```

Fig. 8. An Algorithm for WAE Slicing.

were not in the original program. Finally, the only non-goto nodes in the slice are those in $\psi.S$ which, by construction, contains the same nodes as OttensteinSlice(*P, V*). Therefore, the slice is Ottenstein-equal.

5.2.2. Strong, amorphous, Ottenstein-equal (SAE) slices

To avoid the non-termination potentially present in a WAE slice, the SAE algorithm identifies certain branch nodes from the original program and replaces them with `gotos`. This transformation of branch nodes into goto nodes is termed 'morphing'. The nodes to be morphed are identified by the following definition:

Definition 20. (Morph Branches). For program *P*, set of non-`goto` nodes *S*, and set of goto nodes *G*, Node *b* is an element of MorphBranches(*P, S, G*) iff all of the following hold:

(1) *b* is a branch node of CFG(*P*),
(2) *b* is *not* in *S*,
(3) *b* cannot reach its immediate postdominator by traversing only edges *live* with respect to *S*, and
(4) there exists a node in $S \cup G$ that is an AST descendant of *b*.

An example that includes morph branches appears in Fig. 5. In the centre column, the value of the function MorphBranches(*P*, {5, 6}, {2, 4}) is {1}. In relation to the definition, Node 1 is a branch statement (Condition 1); it is not contained in the set {5, 6} (Condition 2); it cannot reach its postdominator because its `else` branch, which is needed to reach its postdominator, is not live (Condition 3); and Node 2, a member of {5, 6} ∪ {2, 4}, is a descendant of Node 1 (Condition 4).

The first stage of an SAE slice is computed by Function SAE/Slice(*P, V*) (see Fig. 9), which yields the 4-tuple ψ described by the following equations:

$\psi.S$ = OttensteinSlice(*P, V*)
$\psi.G$ = LiveGotos(*P*, $\psi.S$)

```
SAE_Slice(P, V) =
    local ψ, M
    ψ := WAE_Slice(P, V)
    M := MorphBranches(P, ψ.S, ψ.G)
    return (ψ.S, ψ.G, M, ∅)
```

Fig. 9. An Algorithm for SAE slicing.

$\psi.M$ = MorphBranches(*P*, $\psi.S$, $\psi.G$)
$\psi.X = \varnothing$

For essentially the same reason as with the WAE algorithm, the semantic guarantee of an Ottenstein slice [30] holds for those statements represented in $\psi.S$. Furthermore, the slice is strong because the set $\psi.M$ identifies each branch node *b* that, when replaced with `gotos` that target each branch's immediate postdominator *d* (done by Stage 2), ensures that execution passes directly from node *b* to node *d*. This avoids the cycles introduced by Function WAE-Slice, and thus, avoids the possible non-termination introduced by the WAE algorithm. For example, in the center column of Fig. 5, the immediate postdominator of Line 1 is Line 6. Phase 2 replaces Line 1 with the statement '`goto L`', thereby avoiding the cycle present in the WAE slice (shown in the right column). Finally, the sets $\psi.S$ and $\psi.G$ are the same as those computed by WAE/Slice(*P, V*). Thus, an SAE slice is amorphous and Ottenstein-equal for the same reasons that a WAE slice is amorphous and Ottenstein-equal (recall that Ottenstein-equal requires the same number of *non-goto nodes*).

5.2.3. Weak, syntax-preserving, Ottenstein-more (WPM) slices

In order to preserve syntax, implicit `gotos` cannot be made explicit. This is accomplished by including (in $\psi.X$) the branch associated with each implicit `goto`. Inclusion of a branch may make live additional implicit `gotos`, and consequently, this process is iterated until no implicit gotos remain. The required branches are identified by the following definition:

Definition 21. (Exposed Branches). For program *P*, set of non-`goto` nodes *S*, and set of goto nodes *G*, Exposed-Branches (*P, S, G*) = {*b*(*b* ∉ *S* and *b* is AST parent of an *implicit* `goto` node from *G*}.

The program in Fig. 6 motivates the definition of exposed branches. Consider *S* = {1, 3, 10, 11, 12}, the Ottenstein slice of the program taken with respect to the final-use of `x`. In particular, Statement 5 is not in *S* and it controls the

```
WPM_Slice(P, V) =
    local S, G, X, X'
    S := OttensteinSlice(P, V)
    X := ∅
    X' := ∅
    repeat
        X := X ∪ X'
        G := LiveGotos(P, S ∪ X)
        X' := ExposedBranches(P, S ∪ X, G)
    until X' = ∅
    return (S, G, ∅, X)
```

Fig. 10. An Algorithm for WPM Slicing.

implicit `goto` at Statement 8 (*LiveGotos* $=\{2, 4, 6, 8\}$). Therefore, ExposedBranches(P, {1, 3, 10, 11, 12}, {2, 4, 6, 8}) is {5}. Note that '`goto S7`' at Line 9 is not live since the `else`-edge of the '`if (p>q)`' is not live. While no iteration was required for this example program, if Statement 5 of Fig. 6 was nested in an additional condition that could be bypassed via a `goto` statement then iteration would be required.

The first stage of a WPM slice is computed by Function WPM_Slice(P, V) (see Fig. 10), which yields the 4-tuple ψ described by the following equations:

$\psi.S = \text{OttensteinSlice}(P, V)$
$\psi.G = \text{LiveGotos}(P, \psi.S \cup \psi.X)$
$\psi.M = \varnothing$
$\psi.X = \psi.X \cup \text{ExposedBranches}(P, \psi.S \cup \psi.X, \psi.G)$

The equations are recursive since $\psi.G$ and $\psi.X$ are defined in terms of themselves. The algorithm computes the least fixed point solution to these equations. That the least fixed point exists, and consequently the algorithm terminates, follows from the observation that the set of live `gotos` is monotonically increasing and bounded from above, and the following property of exposed branches:

$\text{ExposedBranches}(P, S, G) = X \Rightarrow \text{ExposedBranches}(P, S \cup X, G) = \varnothing$.

Note that the iteration ensures that the solution is minimal, in the sense that no live `goto` so-included could be removed without loss of correctness.

The resulting slice a WPM slice. First, observe that, as with a WAE slice, when the slice and the program terminate, the behaviour of the statements represented in $\psi.S$ is the same in the slice an the Ottenstein slice. Thus, the statements represented by $\psi.S$ have the desired semantics. However, also as with a WAE slice, statements outside ψ.S may introduce non-termination; thus, the slice is weak. Next, observe that there is no morphing required, so the slice is syntax preserving, and finally, by including additional branches the slice is Ottenstein-more.

```
SPM_Slice(P, V) =
    local S, G, M',X'
    S := OttensteinSlice(P, V)
    G := LiveGotos(P, S)
    X' := ExposedBranches(P, S, G)
    M' := MorphBranches(P, S, G)
    while (M' ∪ X') ≠ ∅
        S := OttensteinSlice(P, S ∪ M' ∪ X')
        G := LiveGotos(P, S)
        X' := ExposedBranches(P, S, G)
        M' := MorphBranches(P, S, G)
    end
    return (S, G, ∅, ∅)
```

Fig. 11. An Algorithm for SPM Slicing.

5.2.4. Strong, syntax-preserving, Ottenstein-more (SPM) slices

Like the WPM algorithm, the SPM algorithm iteratively expands the set of statements in the slice. The primary difference is that it includes the Ottenstein slice taken with respect to each exposed branch and morph branch, thereby ensuring that the slice terminates when the original program does. The first stage of an SPM slice is computed by Function SPM_Slice(P, V) (see Fig. 11), which yields the 4-tuple ψ described by the following equations:

$\psi.S = \text{OttensteinSlice}(P, V \cup \psi.S \cup M \cup X)$
where $M = \text{MorphBranches}(P, \psi.S, \psi.G)$, and
$X = \text{ExposedBranches}(P, \psi.S, \psi.G)$
$\psi.G = \text{LiveGotos}(P, \psi.S)$
$\psi.M = \varnothing$
$\psi.X = \varnothing$

Although at first glance the equation for $\psi.S$ may seem incorrect as no union is present in the computation of $\psi.S$, its correctness follows from the equivalence of 'OttensteinSlice(P, $S \cup X$)' and '$S \cup$OttensteinSlice(P, $S \cup X$)'.

Once again, these equations are recursive and the algorithm computes the least fixed point. The existence of least fixed point (and hence termination of the algorithm) follows from the following properties of ExposedBranches and MorphBranches:

$\text{ExposedBranches}(P, S, G) = X \Rightarrow \text{ExposedBranches}(P, S \cup X, G) = \varnothing$, and $\text{MorphBranches}(P, S, G) = M \Rightarrow \text{MorphBranches}(P, S \cup M, G) = \varnothing$

To see that the slice is an SPM slice, first observe that by construction $\psi.S$ equals OttensteinSlice(P, $\psi.S$). Thus, the statements represented in $\psi.S$ inherit the semantic guarantee of an Ottenstein slice including its termination properties. As a result, the slice is strong. Finally, the algorithm is syntax-preserving and Ottenstein-more for the same reasons that the WPM algorithm is syntax-preserving and Ottenstein-more.

5.3. Complexity discussion

In the following discussion, let P represent the size of the program. In addition to the complexity of the two stages described in Section 5, a preprocessing stage (Stage 0) is also considered. Stage 0 builds the immediate postdominator relation, the CFG, the AST, and the PDG. The complexity of these steps is $O(P)$, $O(P)$, $O(P)$, $O(P^2)$, respectively [15,16,28]. The PDG construction complexity is dominated by the computation of data-dependence edges. This cost is highly dependent on precision. For example, a range of pointer analysis with varying complexities exist [22,32].

The complexity of each Stage 1 algorithm is considered separately. For a WAE slice, computing an Ottenstein slice takes $O(P)$ time [28]. Live `gotos` may be computed in $O(P)$ time by traversing the CFG 4 times. The first pass identifies live edges while the second identifies vertices reachable from the nodes of the Ottenstein slice using only live edges. Both of these passes are linear traversals of the CFG. The results are sufficient to identify those live `goto` nodes from the first clause of the definition. For the second clause, two additional traversals are required. The first simply labels nesting level so that the second clause's nesting requirement can be tested in constant time. The final traversal again starts from the nodes of the Ottenstein slice and considers only live edges, but stops at inward `goto` statements that fail the nesting requirement. As each of the four traversals and the Ottenstein slice require linear time, the complexity of Stage 1 of a WAE slice is $O(P)$.

The first two steps of the SAE algorithm are identical to the WAE algorithm; thus, for the same reasons they can be computed in linear time. The definition of MorphBranches has four parts. The first two parts can be checked in constant time. The remaining two can be tested in linear time by precomputing two pieces of information. First, identify the branch nodes that cannot reach their postdominator ignoring `false` edges in the CFG. Using this approach, Condition 3 of Definition 21 can be computed in constant time per branch. (When a branch node is not in the slice, all nodes along all paths to the postdominator, excluding the postdominator, are not in the slice. The `false` edges along paths to the postdominator will not be live when a branch statement is not in the slice.) Second, identify the statements that are not control dependent on their AST parent. Using this, Condition 4 of Definition 21 can be computed in constant time per branch. Thus, the complexity of Stage 1 of an SAE slice is $O(P)$.

For the WPM algorithm, as with the first two algorithms, the set $\psi.S$ can be computed in linear time. At first the computation of live gotos appears more complex as it is computed on each loop iteration. However, the set is monotonically increasing and can be computed incrementally by caching old results. Exposed branches can also be computed in linear time given that the set $\psi.G$ is monotonically increasing: first, the AST parent of new implicit `gotos` added to $\psi.G$ are identified. If a previous implicit `goto` is encountered during this identification, the graph walk may terminate, as further traversal would be redundant. Finally, exposed branches are identified during this walk by checking if branches encountered are in $\psi.S$. As this test takes constant time, the construction of Exposed-Branches and consequently Stage 1 of the WPM slicing algorithm requires $O(P)$ time.

Finally, like WPM, the SPM slicing algorithm involves iteration. Also like WPM, information from previous iterations may be cached or incrementally computed. Thus, the time required to compute Ottenstein Slice, live gotos, exposed branches, and morph branches remains $O(P)$. Combined this yields a complexity for Stage 1 of a SPM algorithm of $O(P)$.

For all four slicing algorithms, Stage 2, the AST projection, takes $O(P)$ time since each node is visited only once. During each visit, the set membership tests can be done in constant time assuming Stage 1 labels each node with its memberships.

6. Classification and comparison with prior work

The problem of slicing unstructured programs is detailed and subtle. There has been a steady study of this problem in the literature [1,2,11,20,25]. In this section existing algorithms are described, placed in the framework introduced in Section 3, and compared to the algorithms presented in Section 5. At this point it is helpful to introduce the following four definitions.

Definition 22. (Language Types). A language is *block-structured* if it has complex statements that are built from other statements, such as `if-then-else` and `while-do` statements. A language that does not have complex statements is a *flat language*. A program is *block-structured* if it is written in a block-structured language and *flat* if it is written in a flat language.

Definition 23. (Disconnected Statements). A statement pair (b, n) is *disconnected* if

(1) b is a branch statement and n is any statement,
(2) b and n are *not* nested (transitively) in different branches of an if statement,
(3) branch statement b appears before statement n in the syntax order, and
(4) there is no executable path from branch statement b to statement n.

Definition 24. (Segmented Program). A program is segmented if it contains a pair of disconnected statements.

For example, the following block-structured program is segmented because the statements '`if(c2)`' and '`i:=i+1`' are disconnected. Condition '`if(c2)`' appears before the statement labeled '`L1`', there is no path from node '`(c2)`' to node '`i:=i+1`', and the two statements are not in different branches of an `if` statement.

```
i:=1;
if (c1) goto L1;
if (c2)
{
  i:=2;
  goto L2;
L1: i:=i+1;
  }
  else
  {
i:=3;
L2: i:=i+2; }
```

For a program to be segmented it does not have to be block-structured. The nesting condition in the definition of a disconnected pair is tautologically satisfied for flat-programs since they have no nesting. Hence, the flat-program equivalent of the above program will also be segmented. Furthermore, as the following example demonstrates, an inward `goto` is not necessary for segmenting a program. The following program is segmented because there is no path from the branch node '`if (c2)`' to the statement labeled '`L1`', although the branch node '`if (c2)`' precedes statement labeled '`L1`' in the syntax order.

```
i:=1;
if (c1) goto L1;
if (c2)
{
  i:=2;
  goto L2;
}
else
{
  i:=3;
  goto L2;
}
L1: i:=i+1;
L2: i:=i+2;
```

Definition 25. (Segmented Slice). An Ottenstein slice is a *segmented slice* if it contains a node n, but does not contain a branch node b, where (b, n) are disconnected.

For example, in the above programs, the Ottenstein slice taken with respect to statement '`i:=i+1`' is segmented since it contains statement '`i:=i+1`', but does not contain branch '`if(c2)`'.

6.1. Classification of prior work

Existing algorithms for slicing in the presence of unstructured control-flow, fall into three of the classifications introduced in Section 3. Section 6.2 compares each of these with the corresponding algorithm from Section 5.

(1) SPM: Ball and Horwitz [2], Choi and Ferrante [11] Algorithm 1, Kumar and Horwitz [25], Agrawal [1];
(2) SAE: Choi and Ferrante [11] Algorithm 2;
(3) WPE: Harman and Danicic [20].

This section uses the three classification dimensions of the paper to classify and compare previous approaches to slicing unstructured programs. As will be seen, previous approaches fall into one of three of the six possible classifications. Previous work was conducted without the benefit of the classifications and so each algorithm is not optimised for the classification in which it resides. The algorithms presented in this paper are optimised for their classification type. This may make the new algorithms superior to those previously published. However, more empirical work is required to assess the impact of this theoretical superiority.

6.2. Comparison with prior work

6.2.1. SPM Algorithms

Ball and Horwitz [2] and Choi and Ferrante [11] independently discovered similar SPM algorithms, referred to as BH and CF1, respectively. Their algorithms identify the statements in a slice by performing a backward traversal of an augmented program dependence graph (APDG). The APDG is constructed like a PDG, but `goto` statements are treated as pseudo predicates. The '`true`' branch of a pseudo predicate leads to the goto target and the '`false`' branch leads to the next statement in the syntax order. BH and CF1 differ on the class of languages to which they apply. BH slices a block-structured language whereas CF1 slices a flat language.

As a result of treating a `goto` statement as a pseudo predicate, both CF1 and BH can introduce spurious control dependences that lead to slices containing non-goto statements not found in the Ottenstein slice. In particular, if the Ottenstein slice is segmented then the CF1 and BH slices will be Ottenstein-more; otherwise they will be Ottenstein-equal. In comparison, the SPM algorithm from Section 5 produces Ottenstein-more slices only when the segmentation is due to an inward `goto`.

More recently, Kumar and Horwitz introduced a new algorithm, referred to as KH, which overcomes the deficiency of BH with regard to spurious control dependences [25]. While KH, like BH, treats goto statements as a pseudo predicates, it does not include the backward slice taken with respect to the pseudo predicates. Unfortunately, KH computes closure (non-executable) slices (it does not consider implicit `goto`s); consequently, its output cannot be used to produce an executable slice by pruning the AST as described in Section 5.1.

Finally, Agrawal's algorithm [1], referred to as Ag, is based on the observation that a `goto` node should be included in a slice if it targets a node other than its own lexical successor. Agrawal's term 'lexical successor' is the 'fall-through statement' of CF1 and the 'continuation node' of BH. Having included a goto node, Ag checks to see if the goto is controlled by a predicate not in the slice. If there is any such predicate p, then p and the slice taken with respect to p are included in the final slice. This step has the effect produced by CF1's use of an augmented CFG. Agrawal's algorithm therefore produces very similar slices to those produced by CF1 and BH.

6.2.2. SAE Algorithms

Choi and Ferrante's [11] Algorithm 2, referred to as CF2, computes SAE slices. It is the only prior algorithm that

produces strong and Ottenstein-equal slices. It achieves Ottenstein equality by replacing each statement that is not in the slice by a `goto` statement that transfers control to the immediate post dominator. The resulting slice is amorphous as it includes `goto` statements not from the original program.

CF2 is similar in spirit to the SAE algorithm from Section 5, in that it first computes the Ottenstein slice and then deletes or replaces with `goto` statements certain nodes not in the slice. The primary difference between the two is that CF2 is not selective in its replacement. It morphs (or 'rewires') all branch nodes that are not in Ottenstein slice. In contrast, the SAE algorithm from Section 5 morphs branch statements *only* when simply deleting a `true` branch will introduce a cycle. Furthermore, it introduces new `goto` statements only when some code in the body of a block (`if/while`) is in the slice, but the `if/while` condition itself is not in the slice. This condition does not arise for flat programs.

The CF2 algorithm addresses only flat programs. For such programs, the SAE algorithm introduces amorphous `goto` statements only when the original program will not terminate when there exist `if` conditions not in the slice and which are replaced by 'true'. This follows from the use of SO(n), the 'true' successor, when computing live goto nodes for unstructured branches not in the slice. The empirical determination of how significant this difference is with respect to CF2 remains a problem for future work.

A direct comparison of SAE and CF2 for block structured programs is not possible because those programs are outside the scope of CF2. A block structured program must first be morphed to a flat program in order to use CF2 for producing a SAE slice.

6.2.3. WPE Algorithms

Harman and Danicic's WPE slicing algorithm, referred to as HD, effectively attempts to transform Agrawal's SPM algorithm to a WPE algorithm. Like the WPE algorithm from Section 5, it does this by first computing an Ottenstein slice and then adding `goto` statements as needed to produce a weak slice. HD is syntax-preserving because it adds only `goto` nodes from the original program. However, it does this at the expense of potentially introducing non-termination and does not account for implicit `goto` nodes.

7. Conclusion and future work

To better understand and classify slicing algorithms for programs with unstructured control-flow, this paper introduces a three-dimensional space; a framework that facilitates exploration of the trade-offs made when slicing unstructured programs. The choice of the most suitable slicing algorithm is impacted by the target application. For example, weak, amorphous, Ottenstein-equal slices may be most acceptable for debugging; strong, amorphous, Ottenstein-equal slices for re-engineering; and strong, syntax-preserving, Ottenstein-more slices for program integration.

Of the eight classifications in the framework, the ideal slice would be a strong, syntax-preserving, and Ottenstein-equal (SPE) slice. Unfortunately, as shown in Section 4, SPE slicers do not exist. Indeed, nor do WPE slicers: even dropping the requirement that the slices be strong (i.e. termination preserving) is insufficient to guarantee existence. Therefore, in general, a slice of an unstructured program must be either Ottenstein-more or amorphous. Furthermore, the existence of WAE and SAE slicers obviates the need for WAM and SAM slicers. This leaves four classifications of interest: WAE, SAE, WPM, SPM.

The paper places existing algorithms for slicing programs with unstructured control flow into these four classifications. It also presents linear time algorithms for the four. In practice, as outlined in Section 6, more work is needed to determine whether there are significant differences in slicing time or slice size. However, the new algorithms are, in theory, an improvement on previous techniques. For example, it is not easy to modify the classification of previous algorithms (e.g. Ball and Horwitz's algorithm is SPM, but cannot easily be modified to give a WPE algorithm). In contrast, the new algorithms all share a common infrastructure; thus, movement between classifications is considerably easier. This facilitates understanding and comparison between the different options available when slicing programs with unstructured control flow.

References

[1] Hiralal Agrawal, On slicing programs with jump statements, in: ACM SIGPLAN Conference on Programming Language Design and Implementation. Orlando, Florida, June 20–24, 1994, pp. 302–312. Proceedings in SIGPLAN Notices, vol. 29(6), June 1994.

[2] Thomas Ball, Susan Horwitz, Slicing programs with arbitrary control-flow in: Peter Fritzson (Ed.), 1st Conference on Automated Algorithmic Debugging, Linköping, Sweden, Springer, 1993, pp. 206–222. (Also available as University of Wisconsin-Madison, technical report (in extended form), TR-1128, December, 1992).

[3] Jon Beck, David Eichmann, Program and interface slicing for reverse engineering, in: IEEE/ACM 15th Conference on Software Engineering (ICSE'93), IEEE Computer Society Press, Los Alamitos, California, USA, 1993, pp. 509–518.

[4] David Wendell Binkley, Precise executable interprocedural slices, ACM Letters on Programming Languages and Systems 2 (1–4) (1993) 31–45.

[5] David Wendell Binkley, Computing amorphous program slices using dependence graphs and a data-flow model, in: ACM Symposium on Applied Computing, The Menger, San Antonio, Texas, USA, ACM Press, New York, NY, USA, 1999, pp. 519–525.

[6] David Wendell Binkley, Keith Brian Gallagher, in: Marvin Zelkowitz (Ed.), Program Slicing Advances in Computing vol. 43, Academic Press, New York, 1996, pp. 1–50.

[7] David Wendell Binkley, Mark Harman, A survey of empirical results on program slicing, Advances in Computers 62 (2004) 105–178.

[8] David Wendell Binkley, Susan Horwitz, Tom Reps, Program integration for languages with procedure calls, ACM Transactions on Software Engineering and Methodology 4 (1) (1995) 3–35.
[9] Gerardo Canfora, Aniello Cimitile, Andrea De Lucia, in: Mark Harman, Keith Gallagher (Eds.), Conditioned program slicing Information and Software Technology Special Issue on Program Slicing vol. 40, Elsevier Science B.V., Amsterdam, 1998, pp. 595–607.
[10] Gerardo Canfora, Aniello Cimitile, Andrea De Lucia, G.A. Di Lucca, Software salvaging based on conditions, in: International Conference on Software Maintenance (ICSM'96), Victoria, Canada, September, IEEE Computer Society Press, Los Alamitos, California, USA, 1994, pp. 424–433.
[11] Jong-Deok Choi, Jeanne Ferrante, Static slicing in the presence of goto statements, ACM Transactions on Programming Languages and Systems 16 (4) (1994) 1097–1113.
[12] Aniello Cimitile, Andrea De Lucia, Malcolm Munro, A specification driven slicing process for identifying reusable functions, Software Maintenance: Research and Practice 8 (1996) 145–178.
[13] Andrea De Lucia, Program slicing: methods and applications, in 1st IEEE International Workshop on Source Code Analysis and Manipulation, Florence, Italy, IEEE Computer Society Press, Los Alamitos, California, USA, 2001, pp. 142–149.
[14] Andrea De Lucia, Anna Rita Fasolino, Malcolm Munro. Understanding function behaviours through program slicing, in 4th IEEE Workshop on Program Comprehension, Berlin, Germany, March. IEEE Computer Society Press, Los Alamitos, California, USA, 1996, pp. 9–18.
[15] Jeanne Ferrante, Karl J. Ottenstein, Joe D. Warren, The program dependence graph and its use in optimization, ACM Transactions on Programming Languages and Systems 9 (3) (1987) 319–349.
[16] Charles N. Fischer, Richard J. LeBlanc, Crafting a Compiler Benjamin/Cummings Series in Computer Science, Benjamin/Cummings Publishing Company, Menlo Park, CA, 1988.
[17] B. Keith Gallagher, James R. Lyle, Using program slicing in software maintenance, IEEE Transactions on Software Engineering 17 (8) (1991) 751–761.
[18] Mark Harman, David Wendell Binkley, Sebastian Danicic, Amorphous program slicing, Journal of Systems and Software 68 (1) (2003) 45–64.
[19] Mark Harman, Sebastian Danicic. Amorphous program slicing, in: 5th IEEE International Workshop on Program Comprenhesion (IWPC'97), Dearborn, Michigan, USA, May, IEEE Computer Society Press, Los Alamitos, California, USA, 1997, pp. 70–79.
[20] Mark Harman, Sebastian Danicic, A new algorithm for slicing unstructured programs, Journal of Software Maintenance and Evolution 10 (6) (1998) 415–441.
[21] Mark Harman, Robert Mark Hierons, An overview of program slicing, Software Focus 2 (3) (2001) 85–92.
[22] Michael Hind, Pointer analysis: haven't we solved this problem yet? in Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01) (Snowbird, Utah, June 18–19, 2001), New York, NY, ACM, 2001, pp. 198–209.
[23] Susan Horwitz, Jan Prins, Thomas Reps, Integrating non-interfering versions of programs, ACM Transactions on Programming Languages and Systems 11 (3) (1989) 345–387.
[24] Mariam Kamkar, Interprocedural dynamic slicing with applications to debugging and testing, PhD Thesis, Department of Computer Science and Information Science, Linköping University, Sweden. Available as Linköping Studies in Science and Technology, Dissertations, Number 297, 1993.
[25] Sumit Kumar, Susan Horwitz, Better slicing of programs with jumps and switches, in: Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering (FASE 2002) of Lecture Notes in Computer Science, vol. 2306, Springer, 2002, pp. 96–112.
[26] Arun Lakhotia, Jean-Christophe Deprez, in: Mark Harman, Keith Gallagher (Eds.), Restructuring Programs by Tucking Statements into Functions, Information and Software Technology Special Issue on Program Slicing vol. 40, Elsevier, 1998, pp. 677–689.
[27] James R. Lyle, Mark Weiser, Automatic program bug location by program slicing, in: 2nd International Conference on Computers and Applications, Peking, IEEE Computer Society Press, Los Alamitos, California, USA, 1987, pp. 877–882.
[28] Karl J. Ottenstein, Linda M. Ottenstein, The program dependence graph in software development environments, SIGPLAN Notices 19 (5) (1984) 177–184.
[29] Lyle Ramshaw, Eliminating goto's while preserving program structure, Journal of the ACM 35 (4) (1988) 893–920.
[30] Thomas Reps, Wuu Wang, The semantics of program slicing and program integration, in: Colloquium on Current Issues in Programming Languages (Barcelona, Spain, March 13–17, 1989), Lecture Notes in Computer Science, Springer-Verlag, New York, NY, vol. 352, 1989, pp. 360–374.
[31] Nahid Shahmehri, Generalized algorithmic debugging. PhD Thesis, Department of Computer Science and Information Science, Linköping University, Sweden. Available as Linköping Studies in Science and Technoloy, Dissertations, Number 260, 1991.
[32] Marc Shapiro, Susan Horwitz, The effects of the precision of pointer analysis, Lecture Notes in Computer Science 1302 (1997) 16–34.
[33] Frank Tip, A survey of program slicing techniques Technical Report CS-R9438, Centrum voor Wiskunde en Informatica, Amsterdam, 1994.
[34] Martin Ward, Program slicing via FermaT transformations, in: 26th IEEE Annual Computer Software and Applications Conference (COMPSAC 2002), Oxford, UK, August, IEEE Computer Society Press, Los Alamitos, California, USA, 2002, pp. 357–362.
[35] Mark Weiser, Program slicing, IEEE Transactions on Software Engineering 10 (4) (1984) 352–357.