The next 700 slicing criteria[*]

Mark Harman, Sebastian Danicic, Yoga Sivagurunathan
Project *Project*,
School of Computing,
University of North London,
Eden Grove, London, N7 8DB.
tel: +44 (0)171 607 2789
fax: +44 (0)171 753 7009
e-mail: `m.harman@unl.ac.uk`

and

Dan Simpson,
Department of Computing,
University of Brighton,
Watts Building,
Moulsecoomb,
Brighton BN2 4GJ.
tel: +44 (0)1273 642451
fax: +44 (0)1273 642405
e-mail: `Dan.Simpson@bton.ac.uk`

2nd Workshop on Program Comprehension, Durham, England, 11th – 12th July 1996

Abstract

A slice is constructed by deleting statements from a program whilst preserving some projection of its semantics. Since Mark Weiser introduced program slicing in 1979, a wide variety of slicing paradigms have been proposed, each of which is based upon a new formulation of the slicing criterion, capturing the semantic projection to be preserved during the process of command deletion.

This paper surveys these slicing criteria, attempting to establish a set of parameters which combine to form a slicing criterion. The effort to abstract a general set of parameters for slicing criteria highlights the existence of many new possibilities for slicing, corresponding to, as yet unpublished, criteria. Many of these novel slicing criteria may find applications in program comprehension and analysis.

The paper introduces no new algorithms for constructing slices, rather it introduces new criteria with respect to which slices might usefully be constructed. The paper also goes some way towards a unification of previous, apparently different, but related, approaches to slicing in which the process of command deletion remains invariant while the semantic projections preserved during the command deletion process vary.

1 Introduction

Many authors [36, 17, 12, 11, 20] have suggested that program slices and their automated computation are helpful to developers and maintainers faced with the problem of understanding programs. Slicing is helpful

[*]The title of this paper is stolen from Peter Landin's 'The next 700 Programming Languages' [24], which begins 'A family of unimplemented computing languages is described that is intended to span differences of application area by a unified framework'. In this paper a family of (implemented and) unimplemented slicing criteria is described that is intended to span differences of application area by a unified framework.

in program comprehension because a slice is a simplified version of the original program, constructed with a specific analysis in mind. Slicing removes parts of the syntax of the program which are not relevant to the analysis in hand, avoiding unnecessary effort on the part of the programmer/maintainer.

The past five years have witnessed a dramatic rise in the volume of literature on program slicing and related source code analysis techniques. This has served to dramatically increase the scope for defining slicing criteria.

In 1991 Venkatesh [34] provided a formal description of program slicing[1], together with a comparative survey of slicing criteria. The present paper could be thought of as an informal, discursive sequel to Venkatesh's paper. Venkatesh draw attention to three orthogonal slicing dimensions, each of which offered a boolean choice. A slice could be static or dynamic, it could be constructed in a forward or backward direction and it could be either an executable program or merely a set of statements related to the slicing criterion.

The present paper identifies four parameters (dimensions in Venkatesh's terminology), the values of which form part of the definition of a slicing criterion. For these four parameters the admissible values are not boolean, but are drawn from a finite set of possible values. A further 'boolean' parameter which controls the choice of whether strict or lazy semantics is to be preserved is also identified. Finally, the direction of the slicing process — forward, backward, either or both is identified as a part of any slicing criterion.

Many of the possible criteria identified by this treatment have yet to be discussed in the literature, let alone implemented. However, the source code for the static and dynamic tools Unravel and Spyder [26, 2] have been made publicly available. This has considerably enriched the potential for practical advances in slicing technology. Recent work has also enriched the variety of criteria with respect to which a slice may be constructed [32, 6, 20]. It is hoped that the wide availability of slicing technology will stimulate development of tools for constructing these new forms of slice.

Slicing, in a very general sense, is a program transformation which preserves some projection of the semantics of the original program. A particular approach to slicing is defined by describing the aspect of the program to be preserved and the nature of the transformations to be performed upon the program to construct a slice. The aspect of the program which must be preserved is captured by the slicing criterion.

Almost without exception[2], the published work on slicing is concerned with a very simple program transformation — that of command deletion. Therefore a less general characterisation of slicing is the process of deleting commands from a program, whilst preserving some aspect of its behaviour as captured by the slicing criterion. It is this characterisation of slicing which shall be adopted throughout the paper. The exception to this comes with the treatment of forward slicing, in which statements are added to an (initially) empty program to form a slice. This is to be expected as the goal adopted in this paper with regard to the formulation of the forward criterion is to provide a 'mirror image' of the backward criterion.

The rest of the paper is organised as follows:-

Section 2 introduces Weiser's original definition of a slice, which is termed a 'static backward slice' in Venkatesh's comparative study, to distinguish it from later developments[3].

The first deviation from Weiser's original formulation of the slicing criterion was Korel and Laski's introduction of the dynamic slice, covered in section 3. As discussed in section 3, Korel and Laski base their slicing criterion on the trajectory induced by the input supplied to a program, and refer not to a point in a *program*, but a point along this *trajectory*. This has the important consequence that the Korel and Laski formulation of dynamic slicing introduces not one, but *two* new slicing criteria parameters — the input sequence and the 'iteration count'. Section 4 isolates the iteration count from Korel and Laski's dynamic slicing criterion, allowing it to be considered independently of any other parameters.

As well as surveying the slicing literature at the time, Venkatesh's paper on the semantics of program slicing

[1]Venkatesh's semantic description was cast in terms of a novel denotational description of a labelled structured language using a concept of contamination. The idea was to capture the set of labels which identify statements and predicates whose computation would become contaminated were some particular variable to be corrupted. Contamination propagates through the semantic description of a program in much the same way that data dependence and control dependence propagate through the Program Dependence Graph (PDG) in slice construction using the PDG approach [30, 18].

[2]A few definitions of slice exist [15, 17, 8], in which a slice need not be a syntactic subset of the program from which it is constructed. The set of transformations admissible for the construction of a slice is another parameter contributing to the definition of program slicing. In the present paper, this will not be considered, as the treatment is already somewhat cluttered by the variety of slicing criteria considered.

[3]This addition of extra adjectives, corresponding to extra parameter instantiations, in the definition of the slicing criterion will, with recent developments, become rather cumbersome. For example, static slicing should, perhaps now, be referred to as 'static, backward, lazy, executable, singleton slice point, singleton iteration count slicing'. The formal lattice theoretical model suggested (though little developed) in the present paper will, the authors hope, go some way to ameliorating this terminological explosion.

[34] introduced a new 'quasi-static' formulation of slicing. A number of recent publications [11, 6, 32, 15] introduce similar, but richer, formulations of 'quasi-static' slicing. These 'quasi-static' criteria are considered in section 5, which shows that quasi-static slicing introduces another orthogonal slicing criterion parameter — the set of initial states for which a slice must preserve a semantic projection of the original program. The inclusion of a set of initial states allows for a definition of a slicing criterion which subsumes the definition of Weiser's static slice, Canfora and De Lucia's conditioned slice[4], Tip's constrained slice and Venkatesh's quasi-static slice, but *not* (due to the presence of the iteration count) of Korel and Laski's dynamic slice.

Sections 6 and 7 briefly review well-known variants on the slicing theme — respectively the concept of a slice as merely a set of statements, (rather than a program whose execution preserves a projection of the original's semantics) and the concept of a slice constructed in the 'forward' (rather than the 'backward') direction. Forward slices, as introduced by Horwitz et al [19], are not executable programs. A natural variant of forward slicing is introduced in section 7, for which forward slices are executable. This is important as it allows forward slicing to be defined as a semantic projection preserving process, revealing it to be a true mirror image of backward slicing.

Until section 8, program semantics will be considered to be lazy. Section 8 briefly reviews the concept of non-termination preservation during slicing (introduced by Kamkar [21]). Following Cartwright and Felleisen's work [7] it is argued that non-Kamkar definitions of slicing preserve a projection of the lazy semantics of the original program, whilst Kamkar-slicing preserves its strict semantics. The issue is closely related to the observation by Cartwright and Felleisen [7] that the implied semantics of a Program Dependence Graph dominate the semantics of the program from which the graph is constructed.

Sections 9 and 10 briefly review end slicing and decomposition slicing, introduced by Lakhotia [23] and Gallagher and Lyle [14].

In section 11 the analyses of the preceding sections are drawn together in a table which identifies the parameters which combine to form many known and novel slicing criteria.

Finally, in section 12, the possibility of simultaneous slicing (slicing with respect to a *set* of slicing criteria) is considered. This possibility allows the approaches described in section 11 to be interwoven to yield an even richer tapestry of slicing criteria. The paper concludes with the suggestion that a formal framework for describing this wide variety of slicing criteria is extremely desirable as a formal tool for managing the explosion in available slicing criteria.

2 Static slicing

The first published definition of a program slicing was given by Mark Weiser, who introduced the concept of program slicing in his 1979 doctoral thesis [35]. His definition of a slice is given in definition 1 below.

Definition 1 (Static Backward Slice) The slicing criterion is a pair (V, n), where V is a set of variables, and n is a point of interest within p. A slice of p is any program which has the same effect as p upon the variables in V at n.

Slicing has many applications. The first to be proposed was the application to the problem of debugging. Consider the example program fragment in figure 1.

The original program in figure 1 is supposed to calculate the sum and product of numbers between one and `n`. The sum is calculated correctly but the product is not. In identifying the bug which causes the program to misbehave, there is no point in considering statements which have no effect upon the final value of the variable `p`, which stores the product. Static slicing on the final value of the variable `p` removes these unnecessary lines from the program thus saving wasted debugging effort.

The important theoretical aspect of the slice w.r.t. $(\{\mathtt{p}\}, 10)$ is the way is preserves, not the conventional semantics of the original program, but a *projection* of its semantics. Rather than preserving the state mapping denoted by the original program (as conventional transformation would typically be required to do), slicing w.r.t. $(\{\mathtt{p}\}, 10)$ preserves the state mapping when states are domain restricted to `p`. The justification for slicing's simplifying power rests upon this 'projection semantics'.

[4]To be precise, the inclusion of a set of initial states does not subsume the definition of a conditioned slice, rather, it is equivalent to it.

1	`scanf("%d",&n);`	1	`scanf("%d",&n);`
2	`s=0;`		
3	`p=0;`	3	`p=0;`
4	`while (n>1)`	4	`while (n>1)`
5	`  {`	5	`  {`
6	`  s=s+n;`		
7	`  p=p*n;`	7	`  p=p*n;`
8	`  n=n-1;`	8	`  n=n-1;`
9	`  }`	9	`  }`
10	`printf("%d",p);`		
Original Program		Slice w.r.t. $(\{\texttt{p}\}, 10)$	

Figure 1: Weiser's Static Slice

1	`scanf("%d",&n);`		
2	`s=0;`		
3	`p=0;`	3	`p=0;`
4	`while (n>1)`		
5	`  {`		
6	`  s=s+n;`		
7	`  p=p*n;`		
8	`  n=n-1;`		
9	`  }`		
10	`printf("%d",p);`		
Original Program		Slice w.r.t. $(\{\texttt{p}\}, 10) < 1 >$	

Figure 2: Dynamic Slicing

3 Dynamic Slicing

Static slices must preserve a projection of the semantics of the original program for *every* possible execution of the program. This tends to yield rather large slices for most 'well written' (i.e. cohesive) programs. This observation was the motivation for work on the use of slicing as the basis for the calculation of *measurements* of program cohesion [4, 29, 27, 25, 28, 31, 16].

However, the original impetus for the development of program slicing arose because of the use of slices in bug-location. This goal, combined with the disappointingly large slices which were constructed from static slicing algorithms gave rise to an interest in dynamic forms of slice.

Korel and Laski [22] were the first to introduce such a dynamic definition of a slice. A dynamic slice need only preserve the effect of the original program upon the slicing criterion when supplied with input x. The dynamic paradigm is ideally suited to bug-location, because a bug is typically detected as the result of the execution of a program with respect to some specific input, rather than by static consideration of the program's properties.

Consider, again, the example in figure 1. Suppose the original program has been executed and the value entered for the variable `n` was 1. The value printed at the end of the execution is incorrect — it is 0 when it should be 1. To locate the bug which causes this error a dynamic slice is constructed (see figure 2). The dynamic slice only identifies those statements which contribute to the value of the variable `p` when the input 1 is supplied to the program. Locating the bug (the faulty initialisation of `p`) in terms of the dynamic slice is thus easier than with either the original program or the corresponding static slice.

This is a rather extreme example of a dynamic slice, because the input causes the `while` loop to be ignored. However, dynamic slicing allows an improvement in precision in several ways. Clearly statements which remain unexecuted are not included in a dynamic slice. In addition, statements which are executed and create data and control dependencies may be removed from the slice should these dependencies be subsequently 'overwritten' during the execution. Also a dynamic slicing algorithm has available more precise information concerning the value of array indexes, allowing a more finely grained analysis of the dependencies which arise as array elements are defined and referenced[5].

A pure definition of dynamic slicing (so-named for reasons which will become clear shortly) is given in definition 2 below.

Definition 2 (Pure Dynamic Slice) A dynamic backward slice of a program p is constructed with respect to a slicing criterion (V, n, x), where V is a set of variables, n is a point of interest within p and x is an input sequence. A dynamic backward slice preserves the projected meaning of p with respect to V at n when supplied with the input x.

Definition 2 is *not* the definition put forward by Korel and Laski. Their definition (called a KL-slice in this paper) is given in definition 3 below:

Definition 3 (KL slice)
Let $C = (x, I^q, V)$ be a slicing criterion of a program P and T a trajectory of P on input x. A *KL dynamic slice* of P on C is any executable program P' that is obtained from P by deleting zero or more statements from it and, when executed on input x, produces a trajectory T' for which there exists an execution position q' such that:
$F(T', q') = DEL(F(T, q), T(i) \notin N' \text{and} 1 \leq i \leq q)$
for all $v \in V$, the value of v before the execution of instruction $T(q)$ in T equals the value of v before the execution of instruction $T'(q')$ in T'
T'(q') = T(q) = I
where N' is a set of instructions in P'.

The definition uses two auxiliary functions on sequences, F and DEL. $F(T, i)$ is the 'front' i elements of T from 1 to i inclusive. $DEL(T, \pi)$ is a filtering operation, which takes a predicate π and returns the sequence obtained by deleting elements of T which satisfy π.

Definition 3 is constructed for a point of interest, n, not in the *program*, but in the *trajectory* of the program induced by the input x. This means that the definition *also* introduces the independent concept of an *iteration count* for which a slice is constructed. The concept of an iteration count is distilled into a separate slicing criterion parameter in the next section.

4 Iteration Count

For one point in the program, there will be zero or more corresponding points in the trajectory of the program induced by some input. Each point in the trajectory corresponds to an execution occurrence of the corresponding point in the program. A new 'iteration-count dependent' definition of a static backward slice is thus possible. This iteration-count slice is defined in definition 4 below. From definition 4 a separate definition of 'dynamic iteration slice' (corresponding to Korel and Laski's definition of 'dynamic slice') can be obtained by taking the union of the definitions of pure dynamic slice and (static) iteration slice.

Definition 4 (Iteration Slice) A (static) iteration slice s of a program p is constructed with respect to a slicing criterion (V, n, i), where V is a set of variables, n is a point of interest within p and i is a natural number. s must preserve the projected meaning of p when execution reaches n for the i^{th} occasion.

Definition 4 can clearly be generalised to cater for a *set* of iteration counts. This yields the 'general iteration slice' (definition 5) below:

Definition 5 (General Iteration Slice) A general (static) iteration slice s of a program p is constructed with respect to a slicing criterion (V, n, I), where V is a set of variables, n is a point of interest within p and I is a set of natural numbers. The slice s must preserve the projected meaning (with respect to V) of p when execution reaches n for the i^{th} occasion where $i \in I$.

[5]This is the motivation for Bell and Munro's use of dynamic analysis to inform static analysis [3].

```
1   while(i<j)
2   {
3   z = y;
4   y = x;
5   x = p;
6   i=i+1;
7   }
```

Figure 3: Iteration Sensitive Dependence

```
1   scanf("%d",&x);
2   scanf("%d",&y);
3   if (x>=0)
4    p = x*y;
5   else
6    p = y*y;
7   printf("%d",p);
```

Figure 4: Quasi Static Slicing

Observe that the conventional definition of static backward slice (definition 1) is obtained from definition 5 but putting $I = \mathbb{N}$, while the definition of a static iteration slice (definition 4) is obtained by putting $I = \{i\}$.

The concept of an iteration number is thus an independent aspect of a program's execution with respect to which slicing criteria may be defined. The iteration concept may be useful in understanding why a program produces incorrect output for some variable on some specific execution of a statement.

Consider, the example in figure 3. On the initial execution of the loop the variable `z` depends upon the initial value of `y` (but does not depend upon the assignments at lines 4 and 5). On the second iteration of the loop the value of `z` does not depend upon the initial value of `y`, but does depend upon the assignment at line 4. By the third iteration (and on all subsequent iterations) the value of `z` depends upon both lines 4 and 5. Slicing with respect to different iteration counts of line number 3 will thus produce different slices. Without the ability to define iteration count(s) in slicing criteria, slicing on line 3 would include the entire program.

5 Quasi–Static and Conditioned Slicing

In order to establish a compromise position between the two extremes of static and dynamic slicing criteria, Venkatesh [34] introduced the concept of a quasi-static criterion. Instead of naming a *specific* input in the slicing criterion, Venkatesh's quasi-static slicing criterion names a *prefix* of the input. Thus a quasi-static slice is constructed with respect to a set of 'possible inputs' to the program, all of which agree on some input prefix. As the length of this prefix may be zero or may be the length of the entire sequence, Venkatesh's quasi-static criterion subsumes both static and dynamic criteria as special cases[6].

Consider the program in figure 4. If the first input value entered is -1 then line 4 will not be executed and therefore need not be in any slice constructed with respect to this partial input information. The enrichment of the criterion to include partial information in this manner is reminiscent of work on partial evaluation of programming languages [5].

Canfora et al [6], introduced definition 6 of slicing which generalises Venkatesh's quasi-static slice to that of a 'conditioned slice', which is constructed with respect to a set of possible input states, characterised by a universally quantified, first order predicate logic formula. The free variables of this formula are a subset of the input variables to the original program. Harman et al [17] introduced a similar definition of a 'general quasi

[6]Venkatesh's original formulation [34] involves slicing 'at the end of the program' (what Lakhotia [23] terms 'end slicing' (see section 9)). Strictly speaking, therefore, Venkatesh's quasi-static slicing subsumes neither static nor dynamic slicing, although an obvious generalisation does.

```
1   scanf("%d",&x);
2   scanf("%d",&y);
3   if(x>y)
4     z = 1;
5   else
6     z = 2;
7   printf("%d",z);
```

Figure 5: Conditioned slicing

static' slice, for which construction is not limited to command deletion. Tip [32] introduced a term rewriting algorithm which constructs a more restricted[7] form of conditioned slice, called a 'constrained slice'. For a constrained slice the set of possible input states is characterised by a predicate in the programming language[8].

Definition 6 (Conditioned Slice) A conditioned slice of a program p is constructed with respect to a quadruple (I, n, π, V), where I is a set of variable names whose value is obtained from the input sequence, n is a point of interest within p, π is a predicate whose free variables are a subset of I and V is a set of variable names. A conditioned slice must preserve the projected meaning (with respect to V) of p, when executed in an initial state satisfying π.

Consider the example program in figure 5. If the program is executed in a state in which the value read into `x` is greater than the value read into `y` then line 6 will not be executed and therefore need not appear in any slice constructed with respect to such a set of initial states.

Conditioned slicing is a significant advance on both static and dynamic slicing, as it subsumes both as special cases, in addition to making possible a wide spectrum of intermediate possibilities. Conditioned slicing also subsumes Venkatesh's concept of a quasi-static slice.

Specifically:

- To capture the pure dynamic slicing criterion (definition 2), the predicate mentioned in the conditioned slicing criterion will equate a value with each variable[9] — that is, the value that is assigned from the input sequence x.
- To capture the static slicing criterion the predicate mentioned in the conditioned slicing criterion will simply be the constant nullary predicate 'true', since a static slice must preserve the projected meaning of the original program regardless of the initial state.
- To capture Venkatesh's quasi-static criterion the predicate mentioned in the conditioned slicing criterion should equate a value with each variable assigned a value from the input prefix mentioned in the quasi-static slicing criterion.

For example, the conditioned slicing criterion can capture the fact that the initial value of `x` is -1, thus the conditioned slice of the example in figure 4 can be used in place of the quasi-static slice constructed for an input sequence which begins with the value -1. Moreover, the conditioned criterion could capture the more general situation in which the initial value of `x` is simply some negative number. This captures not one, but all, of the cases in which line 4 will fail to be executed. Also, observe that the quasi-static criterion cannot capture the situation where `x` is greater than `y` (in which case the example in figure 5 will fail to execute statement 6). The best that could be achieved in such a situation would be to specify both the initial value of `x` *and* that of `y`, in which case quasi-static slicing would simply degenerate to dynamic slicing.

[7]The current implementation of constrained slicing allows equality and relational operators in 'constraints'. Constraints play an identical role to conditions in conditioned slicing. However, this is an implementation decision — the definition of constrained slicing, can, in theory, allow for an arbitrary set of initial states [33].

[8]To be precise, Canfora et al [6] introduce conditioned slicing in terms of a condition from the programming language notation, whereas De Lucia [11] generalises this condition to be an arbitrary, universally quantified, formula of first order predicate logic.

[9]De Lucia [11] uses subscripts to denote different values stored in the same variable location during different iterations of a loop.

Observe that conditioned slicing cannot, however, capture Korel and Laski's definition of a dynamic slice (definition 3), as this definition involves the concept of an iteration count. Clearly however, it would be possible to augment the conditioned slicing criterion with an extra iteration count parameter, thereby allowing it to subsume the definition of dynamic slicing introduced by Korel and Laski. It is not presently obvious what practical ramifications for conditioned slicing algorithm design would result from such an augmentation of the conditioned slicing criterion.

Finally, it is worth noting that the conditioned criterion need not be constructed with respect to a set of inputs, but could, more generally, be constructed with respect to a set of initial *states*. The would allow the technique to be applied to arbitrary code fragments in which some variables are 'uninitialised'.

6 Non–Executable slices

Agrawal and Horgan [1] and Horwitz et al [18] introduce the concept of a slice as a set of statements for the dynamic and static slicing paradigms respectively. These forms of slice are *not* executable programs and, as such, it is not possible to capture the semantic properties of a slice in terms of the equivalence preserved by the slicing process with respect to the original program. Such definitions of slicing are thus beyond the scope of the present treatment.

7 Forward Slicing

Hitherto in this paper, all definitions of slicing considered have been 'backward' definitions of slice. Horwitz et al [19] introduced the concept of a forward slice. As introduced in [18] a forward slice consists of the set of statements affected by the slicing criterion (whereas a backward slice consists of those statements which affect the slicing criterion). This is the conventionally accepted definition of forward slicing taken from [19], and repeated in definition 7 below:

Definition 7 (Conventional Forward Slice)
A forward slice is constructed from a program, p, with respect to a pair (V, i). It is the set of statements and predicates which are affected by the values of any of the variables in V at i in p

However, as indicated in the previous section, the conventional definition of a forward slice leads to slices which are 'not executable', placing the slices so-defined beyond the scope of the present treatment. Fortunately, it is possible to define a form of 'forward' slicing which is faithful to the spirit of forward slicing and for which the slices constructed will be executable programs.

Instead of defining of forward slice to contain those components of the program which are *affected by* the slicing criterion, it is possible to define a forward slice to be a program whose execution is *unaffected by* the slicing criterion. The 'unaffected' version is simply the complement of the set of statements in the forward slice as given by definition 7.

Consider the example program in figure 6. Forward slicing with respect to the criterion $(1, \{\mathtt{y}\})$ will reveal that the criterion affects line 5 (by forward data flow) and line 6 (by forward control flow). According to definition 7 of a forward slice, the slice therefore consists of lines 5 and 6 alone. Although this slice is a program which can be executed, it is termed a 'non-executable slice' because it cannot be related to the execution of the original program from which it is constructed.

By contrast, the complement of the traditional forward slice (those lines which are definitely not affected by the slicing criterion) consists of lines 1,2,3,4 and 7. The lines form an executable program which *is* related to the original because it produces the same final state when both are executed in initial states projected onto the complement of the variables of the slicing criterion.

This definition of forward slice seems to be, at least theoretically, a more natural mirror image of backward slicing, since both forward and backward slicing will be executable programs, related to the original program by the manner in which they preserve properties of its execution.

Definition 8 (Executable Forward Slice) An executable (static) forward slice is constructed with respect to a slicing criterion, (V, n) by adding statement to an (initially) empty program. The forward slice must have the same meaning as the original program when the state at n is projected onto V.

```
1   x=y;
2   if (p<q)
3   {
4     z=z+1;
5     if (y<q)
6       z=z-1;
7   }
```

Figure 6: Forward slicing

```
1   while(y!=N)
2     y=y+1;
3   x=1;
4
```

Figure 7: Non-Termination Preservation

Whereas the backward slice is constructed by deleting commands from the original to form a slice, the forward slice is constructed by adding commands from the original to an (initially) empty program. The best backward slice is the smallest program which preserves the effect of the original program upon the slicing criterion, while the best forward slice in the largest program upon which the complement of the slicing criterion has no effect.

This appears (at least to the authors) to capture the desired property of a forward slice — namely that it mirrors the definition of a backward slice. It should ne possible, with this, slightly altered, definition of a forward slice, to define a forward slice in terms of a set of corresponding backward slices.

Finally, as the new definition of forward slice proposed here is simply the complement of the original definition proposed by Horwitz et al, it would clearly be possible to maintain the original 'set of statements' view of the contents of the forward slice, and merely and an extra complement operation to the definition.

8 Non–Termination preservation

All definitions of slicing considered in this paper have been constructed with respect to the lazy semantics of the original program from which they are constructed. That is, the remark that a slice 'preserves a projection of the semantics of the original program' should be taken to mean that a slice 'preserves a projection of the *lazy* semantics of the original program'. This has the semantically interesting consequence that slicing may *introduce termination* (although it shall never *introduce non-termination*). Whilst lazy semantics is the norm for functional programming languages, it is not normally associated with the meaning of imperative programs, for which slicing is, almost exclusively, applied. The justification for this lazy perspective on the meaning of imperative programs is provided by Cartwright and Felleisen [7], who show that the semantics of program dependence graphs (from which slices are typically calculated) dominate the conventional semantics of the programs they represent.

Consider the example program in figure 7. A static slice constructed with respect to $(\texttt{x}, 4)$ will (conventionally) contain line 4 alone. The fact that line four will never be executed when `y` is initially greater than `N` is of no consequence. In the lazy interpretation of the program's semantics one could imagine that the program is executed in parallel with a single processor for each variable, and with each processor only executing the statements necessary to define the value of its associated variable. The program may not terminate for the value of `y`, but it definitely does (under this interpretation) for the value of `x`.

Using Kamkar's definition of control dependence, line three will depend (by control flow) on line 1, and thus the slice on `x` will be the entire program.

The reason why slicing rests upon a lazy interpretation of imperative languages, derives from the conventional definition of control dependence [13]. According to this definition a predicate p controls a predicate or statement

n iff the evaluation of p 'decides' whether or not n is either definitely executed or possibly ignored. If such a predicate p controls the execution of a loop which fails to terminate then p only controls those statements in the body of the loop[10]. In particular, if n post dominates p then it is *not* controlled by p, even in situations where the evaluation of p may lead to non-termination.

Kamkar [21] introduces a definition of termination-preserving control dependence which contains the conventional control dependence relation. In addition to the conventional relation, in the termination-preserving control dependence relation a predicate p controls any node n which may fail to be executed as the result of a potentially non-terminating loop controlled by p.

The projection semantics to be preserved by slicing is another parameter to the definition of the slicing criterion. Using the conventional definition of control dependence, slicing will preserve a projection of the *lazy* semantics of the original program — slicing may introduce termination. Using Kamkar's definition of control dependence, slicing will preserve a projection of the *strict* semantics of the original program — a slice will respect the termination characteristics of the original.

The strict version of slicing will always create slices which are at least as 'thick' as the lazy version, which perhaps explains the relative lack of interest in this novel definition of control dependence within the context of slicing.

9 End Slicing

Lakhotia [23] introduces the concept of an end slice, based upon the static backward slice. An end slice is constructed simply with respect to a set of variables V. The point of interest in the original program, at which the slice is constructed, is derived from the program to be sliced. This point is simply the 'end' of the program. The aim is that an end slice captures the program components which contribute to the final value of the set of variables V.

Some care is required in the definition of the 'end of the program' however as, strictly speaking, a slice at the last line of the program cannot, by definition[11] include the last line. This issue highlights the rather awkward nature of describing slices with respect to *programs* and to 'points within programs', as the algorithms for slicing are all based upon the *Control Flow Graph* (CFG) or the *Program Dependence Graph* (PDG) of the program to be sliced. This issue of the point at which to construct a slice in order to capture the final value of a variable disappears in both the CFG and the PDG, which has an extra exit node, which is the last node on every path through the graph. The phrase the 'end of the program' thus refers to the exit node of the program's graph.

10 Decomposition Slicing

Gallagher and Lyle [14] introduce the concept of a decomposition slice, which captures all computation upon a chosen variable v. This is achieved by slicing the program at all points at which the variable v is output and also at the end of the program. Thus the point at which the slice is constructed is no longer required in the slicing criterion, rather it is calculated.

Definition 9 (Decomposition Slice) A decomposition slice s of a program p is constructed with respect to a variable v. It consists of the union of the static backward slices constructed for the criteria $(\{v\}, \mathit{end})$ together with $\{(\{v\}, n_1), \ldots, (\{v\}, n_m)\}$, where $\{n_1, \ldots, n_m\}$ is the set of lines at which v is output in p and *end* is the 'end' of the program p.

Observe that a decomposition slice contains an end slice.

According to definition 9 above, the decomposition slice could be viewed as a special case of simultaneous slicing, in which the set of slicing criteria are derived from the program and the variable of interest (see section 12).

Alternatively, decomposition slicing can be viewed as a variation on Weiser's original definition of a slice (definition 1 in this paper). Whereas Weiser's slicing criterion contains a set of variables and a single slice point,

[10]For unstructured programs the 'body' of a loop may be harder to describe, but the 'laziness' of the semantic projection which is preserved remains the same.

[11]as preservation of the values of a set of variables *at* a point n in a program is taken to mean their preservation *immediately before* the execution of the corresponding node n in the CFG of the program.

the decomposition slice contains a set of points of interest (calculated from the program to be sliced) and a single variable.

This suggests the possibility of slicing not merely at a single point within the original program, but at an arbitrary number of points within the program, revealing that the conventional decision that the slice point is a single statement can easily be generalised. This observation yields the generalisation of Weiser's static slice given in definition 10 below:

Definition 10 (Multi-Point Slice) A multi-point slice s of a program p is constructed with respect to a pair (V, N), where N is a set of points within p and V is a set of variables. A slice must have the same effect as p on all variables in V when a statement at any of the points in N is executed.

The decomposition slice can now be re-formulated in terms of multi-point slicing by definition 11 below:

Definition 11 (Decomposition Slice) A decomposition slice s of a program p is constructed with respect to a variable v. It is the static, multi-point, slice on $(\{v\}, \{n_1, \ldots, n_m\} \cup \{end\})$, where $\{n_1, \ldots, n_m\}$ is the set of lines at which v is output in p and end is the exit node of p.

11 The Variety of Slicing Criteria

The rows of the table in figure 8 set out some of the existing slicing criteria and some of the possible novel definitions of slicing criteria. The columns identify the parameters which define each criterion.

The set I refers to the set of all program variables, the set S to the set of all possible initial states. The corresponding columns of the table show the set of possible values admissible for each parameter, thus $\{\{1\}\}$ in the column for iteration counts indicates that the slice only preserves a projection of the semantics of the original program at the first iteration of the slice point(s). This is written $\{\{1\}\}$ and not $\{1\}$ as it is possible, for example, to have an entry of the form $\{\{3\}, \{2, 6\}, \{n \mid n > 30\}\}$, indicating that the admissible choices of iteration count are either 3 counts only, both 2 and 6 counts or all counts over 30. Similarly, $\{S\}$ in the column for initial states indicates that, in all slicing criteria, all possible initial states are 'of interest'. This is simply a characterisation of static slicing.

The entry 'singleton' means that any single element of the appropriate set may be used, but that combinations are not allowed. For example, in the row for Weiser Static slicing, only a singleton set of slice points may be mentioned in the slicing criterion. There is freedom to choose the point of interest, but only one point of interest per slicing criterion is permitted.

The entry 'derived singleton' means that only one element is permitted and that there is no choice as to what that element is — its value is derived from the original program to be sliced. For example, the entry 'derived singleton' for the slice points in the row for Lakhotia End slicing indicates that only one point is permitted and that this value is derived — in this case it is the exit node of the original program's CFG.

The entry 'derived' indicates that a non–singleton set is possible, but that this set is derived from the original program, once again, allowing no choice. This is the case with Gallagher and Lyle's Decomposition Slice, for which the set of points of interest are calculated to be the points at which the variable of interest is output in the original program together with the 'end' of the original program.

The entries for the direction in which the slice is constructed are F for forward, B for backward, $F \vee B$ meaning either forward or backward (or both) and $F \wedge B$ meaning both forward and backward.

11.1 Novel Slicing Criteria

Many of the entries in figure 8 represent unpublished criteria. There many other criteria which have been omitted for brevity.

The most 'flexible' form of slicing (General Slicing) is that which allows a free choice in all parameters. This definition subsumes all the other definitions, but is really *so* general that it may not be helpful in practice. By contrast the 'Redundant Code' slicing criterion allows no choice in any of the possible parameters. For this definition a statement may only be removed if it has no effect upon any variable at any statement, on any iteration of the statement in either direction for any set of initial states. Clearly, for such a criterion the only statements which may be legally removed are those which can have little bearing[12] on the behaviour of the program. Slicing according to this criterion thus reduces to redundant code removal.

The General and the Redundant Code criteria form the upper and lower bounds of a lattice of potential criteria ordered elementwise by set inclusion on the first four columns of the table. In this ordering, backward and forward slicing are incomparable, but $\vee$ acts as a least upper bound operator and $\wedge$ as a greatest lower bound operator in the lattice. With further work, the size of this lattice may increase due to the inclusion of extra slicing parameters. The goal of this paper is to encourage research into the nature of this lattice and the criteria slicing criteria to be found within it.

The 'conditioned program', as introduced by Canfora and De Lucia [6, 11], was not originally regarded as a form of slice. However, the analysis presented in this paper suggests that it could be by appropriate choice of parameters. Were it not for the set of initial states in the criterion for constructing a 'conditioned program' it would degenerate to the definition of a 'Redundant Code' slice. The algorithm described by De Lucia for constructing a conditioned program with respect to a set of states Σ, only removes statements which are not reachable when the program is executed in any state in Σ. In particular, it does not remove statements which never affect any variables (i.e. redundant code), but which are reachable from some state in Σ [10]. However, this algorithm for constructing conditioned programs calculates imprecise (but correct) slices in terms of the view, adopted here, of a conditioned program as a form of slice.

The three unpublished 'forward' criteria are simply forward versions of popular backward slicing criteria. The reason for drawing attention to them lies in their potential application to program comprehension (and subsequent maintenance activities). Using these forward slicing criteria, a programmer will be able to ascertain the potential effect of variables in a variety of initial starting conditions.

The naïveté referred to in the table is a reflection of the fact that for simultaneous slicing, *different sets* of variables may be associated with *different points* in the program, and perhaps, even with different iteration counts. Multi-point slicing is thus a much reduced form of simultaneous slicing, in which the same set of variables, of iteration counts and of initial states are associated with each point in the program. This observation leads one to believe that, perhaps even this attempt at a general treatment of slicing criteria, is too restrictive. Perhaps the various parameters available should not be independent. This issue is discussed in more detail in section 12.

The variable oriented slicing criterion is aimed at the comprehension of the overall computation upon a single variable at some point in the program, and thus bears a similarity to the Decomposition Slice of Gallagher and Lyle [14]. The variable oriented slice is, however, different to the Decomposition Slice in that it is constructed for a single point in the program (not a derived set of points as with the decomposition slice) and it is constructed for both its effect on the program that contains it (forward slicing) and for the effect the program that contains it has upon it (backward slicing). Variable oriented slicing provides information about the 'significance' of a variable in a program [16], it captures all the statements which contribute to the value of the variable and all those which will be affected by it at the chosen point in the program. This may be of use in program comprehension because it provides a static snapshot of a variable's possible pasts and futures. A natural generalisation of variable oriented slicing would be to add a set of states to the criterion which would allow for the kind of specialisation performed by Canfora's, De Lucia's and Tip's algorithms for conditioned/constrained slicing [6, 11, 32].

As previously observed, the iteration count relating to a point of interest within a program provides a useful tool for specialising slices focused upon debugging effort. Often bugs manifest themselves, not for all possible executions of a statement of interest, but rather they only occur on specific executions of a statement of interest. The ability to specialise a program (using slicing) with respect to a specific iteration counts may also be useful in program comprehension as it allows additional limits to be imposed upon the aspects of the program which are 'of interest'.

12 Simultaneous Slicing

Some algorithms for slicing [19, 30, 9, 11] allow static slices to be constructed with respect to a *set* of slicing criteria. Rather than mentioning a *single* point of interest within the original program, such definitions mention a set of points of interest, each of which is associated with its own set of variables of interest. A simultaneous

[12]Here there remains the issue of the choice of strict or lazy semantics. In the lazy version of slicing, a `while` loop with an empty body and constant predicate 'true' will be removed by redundant code slicing, but will remain in the strict counterpart.

Slicing Paradigm	Variables	Initial States	Slice points	Iteration Counts	Direction
Weiser Static	$\mathcal{P}(I)$	$\{S\}$	singleton	$\{\mathbb{N}\}$	B
Korel and Laski Dynamic	$\mathcal{P}(I)$	all agree on input	singleton	singleton	B
Horwitz et al Forward	$\mathcal{P}(I)$	$\{S\}$	singleton	$\{\mathbb{N}\}$	F
Venkatesh Quasi Static	$\mathcal{P}(I)$	all agree on input prefix	derived singleton	$\{\{1\}\}$	B
Delucia, Cimitile, Tip Conditioned/Constrained	$\mathcal{P}(I)$	$\mathcal{P}(S)$	singleton	$\{\mathbb{N}\}$	B
Gallagher and Lyle Decomposition	singleton	$\{S\}$	derived	$\{\mathbb{N}\}$	B
Lakhotia End	$\mathcal{P}(I)$	$\{S\}$	derived singleton	$\{\{1\}\}$	B
De Lucia Conditioned Program	$\{I\}$	$\mathcal{P}(S)$	$\{\mathbb{N}\}$	$\{\mathbb{N}\}$	$B \wedge F$
Unpublished Criteria					
General	$\mathcal{P}(I)$	$\mathcal{P}(S)$	$\mathcal{P}(\mathbb{N})$	$\mathcal{P}(\mathbb{N})$	$B \vee F$
Forward Pure Dynamic	$\mathcal{P}(I)$	all agree on input	singleton	$\{\mathbb{N}\}$	F
Forward KL Dynamic	$\mathcal{P}(I)$	all agree on input	singleton	singleton	F
Forward Conditioned	$\mathcal{P}(I)$	$\mathcal{P}(S)$	singleton	$\{\mathbb{N}\}$	F
Pure Dynamic	$\mathcal{P}(I)$	all agree on input	singleton	$\{\mathbb{N}\}$	B
‘Lakhotia’ Start	$\mathcal{P}(I)$	$\{S\}$	$\{\{1\}\}$	$\{\mathbb{N}\}$	F
Naïve Multi	$\mathcal{P}(I)$	$\{S\}$	$\mathcal{P}(\mathbb{N})$	$\{\mathbb{N}\}$	B
Naive Iteration	$\mathcal{P}(I)$	$\{S\}$	singleton	$\mathcal{P}(\mathbb{N})$	B
Variable Oriented	singleton	$\{S\}$	singleton	$\{\mathbb{N}\}$	$B \wedge F$
Redundant Code	$\{I\}$	$\{S\}$	$\{\mathbb{N}\}$	$\{\mathbb{N}\}$	$B \wedge F$

Figure 8: A Few of the Variety of Slicing Criteria

static slice is thus a generalisation of definition 1 of static slice (in which the set of slicing criteria is simply a singleton set). However, the concept of simultaneous slicing is also applicable to the other forms of slicing described in this paper.

More importantly perhaps, the possibility of choosing *different* values for criteria parameters for different points within the program opens up considerable potential for sophisticated analyses which may provide assistance in program comprehension. The introduction of various forms of quasi-static slice by Canfora, De Lucia at al, Tip and Venkatesh represents a significant generalisation of slicing in itself. When combined with the concept of simultaneous slicing the approach is further generalised because the set of states identified in these quasi-static criteria can be varied from point to point within the original program. As an example of this generality, consider the general, simultaneous slicing criterion below:

$$\{(\{x\}, \{\sigma \mid \sigma(y) > \sigma(x)\}, \{4\}, \{n \mid n \bmod 2 = 1\}), (\{y, z\}, \{\sigma \mid \sigma(y) = \sigma(x)\}, \{10, 17\}, \mathbb{N})\}$$

This simultaneous general criterion contains two slicing criteria, each of which captures an aspect of the semantics of the original program which any slice must preserve. The first criterion requires that the slice preserves the value of the variable x at statement number 4 if executed in a state where the value of the variable y is greater than that of x and statement number 4 is being executed for the n^{th} occasion where n is an odd number. The second criterion requires that the slice preserves the value of the both the variable y and the variable z on any occasion when execution reaches statement number 10 or statement number 17 after execution in an initial state in which the value of y is identical to that of x.

The technology for constructing such general simultaneous slices appears to be well within reach; the problem of constructing slices with respect to a set of slicing criteria simply reduces to the problem of constructing the union of the slices for each individual criterion [19, 9]. However, it should be pointed out that the problem of constructing slices for particular iteration counts has only been considered in the dynamic paradigm [22], and its extension to the quasi-static paradigm may therefore introduce new problems.

13 Conclusion and Future Work

This paper introduces an informal framework in which slicing criteria may be compared and from which new criteria may be constructed in terms of several 'slicing criteria parameters'.

Specifically, it is argued that the Korel and Laski's definition of dynamic slicing [22] introduces a separate concept of iteration count in the construction of slicing criteria, that forward slices can be defined as executable programs, that conditional slicing subsumes pure dynamic, quasi-static and constrained slicing and that the choice of slice 'direction' and lazy/strict semantic preservation all contribute to the variety of slicing criteria which may used to specialise programs. The paper also shows, by example, that the concept of simultaneous slicing can be applied to all the parameters drawn out by this analysis, and introduces a general, possibly too general, definition of a slicing criterion which subsumes many existing slicing criteria.

The parameters involved in the definition of a slicing criterion identified in this paper form a lattice containing many, perhaps even all, well known slicing criteria as well as a wide variety of new criteria which may be useful in source code analysis and comprehension. A formal lattice theoretical model of slicing criteria would be attractive as it would provide a unified framework for comparing approaches to slicing, independent of the algorithms used to construct them. Such a formal framework remains a problem for future work.

The present paper makes no attempt to define algorithms for constructing slices for the new slicing criteria proposed (although many may be constructed using simple extensions of existing slicing technology). Clearly, some of the criteria proposed may turn out to be unimportant. However, the authors believe that the novel criteria which arise from analysis presented here are likely to find fruitful application in program comprehension and related analyses.

References

[1] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 246–256, New York, June 1990.

[2] Hiralwal Agrawal, Richard A. DeMillo, Hsin Pan, Eugene H. Spafford, and Chonchanok Viravan. Spyder project.

[3] Gordon Bell and Malcolm Munro. Using dynamic analysis to improve static analysis. In Malcolm Munro, editor, 2^{nd} *UK Workshop on Program Comprehension*, Durham, UK, July 1996.

[4] James M. Bieman and Linda M. Ott. Measuring functional cohesion. *IEEE Transactions on Software Engineering*, 20(8):644–657, August 1994.

[5] D. Bjørner, Andrei P. Ershov, and Neil D. Jones. *Partial evaluation and mixed computation.* North–Holland, 1987.

[6] G. Canfora, A. Cimitile, Andrea De Lucia, and G. A. Di Lucca. Software salvaging based on conditions. In *International Conference on Software Maintenance (ICSM'96)*, pages 424–433, Victoria, Canada, September 1994. IEEE.

[7] Robert Cartwright and Matthias Felleisen. The semantics of program dependence. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–27, 1989.

[8] Jong–Deok Choi and Jeanne Ferrante. Static slicing in the presence of goto statements. *ACM Transactions on Programming Languages and Systems*, 16(4):1097–1113, July 1994.

[9] Sebastian Danicic, Mark Harman, and Yogasundary Sivagurunathan. A parallel algorithm for static program slicing. *Information Processing Letters*, 56(6):307–313, December 1995.

[10] Andrea De Lucia. Private communication, 1996.

[11] Andrea. De Lucia, Anna Rita Fasolino, and Malcolm Munro. Understanding function behaviors through program slicing. In 4^{th} *IEEE Workshop on Program Comprehension*, Berlin, Germany, March 1996.

[12] Andrea De Lucia and Malcolm Munro. Program comprehension in a reuse reengineering environment. In Malcolm Munro, editor, 1^{st} *Durham workshop on program comprehension*, Durham University, UK, July 1995.

[13] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9:319–349, July 1987.

[14] Keith B. Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, August 1991.

[15] Mark Harman and Sebastian Danicic. Using program slicing to simplify testing. *Journal of Software Testing, Verification and Reliability*, 5:143–162, September 1995.

[16] Mark Harman and Sebastian Danicic. Towards the measurement of objects. In Martin Shepperd, editor, 1^{st} *Bournemouth Metrics Workshop*, Bournemouth University, UK, April 1996.

[17] Mark Harman, Sebastian Danicic, and Yogasundary Sivagurunathan. Program comprehension assisted by slicing and transformation. In Malcolm Munro, editor, 1^{st} *Durham workshop on program comprehension*, Durham University, UK, July 1995.

[18] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 25–46, Atlanta, Georgia, June 1988. Proceedings in *SIGPLAN Notices*, 23(7), pp.35–46, 1988.

[19] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12:26–61, 1990. Extended version of [18].

[20] Daniel Jackson and Eugene J. Rollins. Chopping: A generalisation of slicing. Technical Report CMU-CS-94-169, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, July 1994.

[21] Mariam Kamkar. *Interprocedural dynamic slicing with applications to debugging and testing.* PhD Thesis, Department of Computer Science and Information Science, Linköping University, Sweden, 1993. Available as Linköping Studies in Science and Technology, Dissertations, Number 297.

[22] Bogdan Korel and Janusz Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, October 1988.

[23] Arun Lakhotia. Rule–based approach to computing module cohesion. In *Proceedings of the* 15^{th} *Conference on Software Engineering (ICSE-15)*, pages 34–44, 1993.

[24] Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, March 1966.

[25] H. D. Longworth, L. M. Ott, and M. R. Smith. The relationship between program complexity and slice complexity during debugging tasks. In *Proceedings of the Computer Software and Applications Conference (COMPSAC'86)*, pages 383–389, 1986.

[26] James R. Lyle, Dolores R. Wallace, James R. Graham, Keith B. Gallagher, Joseph P. Poole, and David W. Binkley. Unravel project.

[27] L. M. Ott and J. J. Thuss. Slice based metrics for estimating cohesion. In *Proceedings of the IEEE-CS International Metrics Symposium*, pages 78–81, 1993.

[28] Linda M. Ott. Using slice profiles and metrics during software maintenance. In *Proceedings of the* 10^{th} *Annual Software Reliability Symposium*, pages 16–23, 1992.

[29] Linda M. Ott and J. J. Thuss. The relationship between slices and module cohesion. In *Proceedings of the* 11^{th} *ACM conference on Software Engineering*, pages 198–204, May 1989.

[30] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in software development environments. *SIGPLAN Notices*, 19(5):177–184, 1984.

[31] J. J. Thuss. An investigation into slice–based cohesion metrics. Master's thesis, Michigan Technological University, 1988.

[32] Frank Tip. *Generation of Program Analysis Tools.* PhD thesis, Centrum voor Wiskunde en Informatica, Amsterdam, 1995.

[33] Frank Tip. Private communication, 1996.

[34] G. A. Venkatesh. The semantic approach to program slicing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 26–28, Toronto, Canada, June 1991. Proceedings in *SIGPLAN Notices*, 26(6), pp.107–119, 1991.

[35] Mark Weiser. *Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method.* PhD thesis, University of Michigan, Ann Arbor, MI, 1979.

[36] Mark Weiser. Programmers use slicing when debugging. *Communications of the ACM*, 25(7):446–452, July 1982.