Amorphous Program Slicing*

Mark Harman and Sebastian Danicic
Project *Project* ,
School of Computing,
University of North London,
Eden Grove, London, N7 8DB,
United Kingdom.
tel: +44 (0)171 607 2789,
fax: +44 (0)171 753 7009.
m.harman@unl.ac.uk, http://www.unl.ac.uk/~mark/welcome.html

Keywords: Slicing, Transformation, Program Specialisation

Abstract

This paper introduces amorphous program slicing. Like traditional slicing, amorphous slicing simplifies a program while preserving a projection of its semantics. Unlike traditional slicing, amorphous slicing may make use of any simplifying transformation which preserves this semantic projection, thereby improving upon the simplification power of traditional slicing and consequently its applicability to program comprehension.

The paper also introduces a theoretical framework of program projection. A projection is defined with respect to an equivalence relation on programs together with a simplicity measure (an ordering on programs). Having defined this framework, amorphous and traditional forms of static and conditioned slice are defined by instantiating the definition of a projection with different equivalence and ordering relations. The projection framework helps to contain the potential explosion in slicing paradigms and facilitates comparison across the boundaries of these paradigms.

1 Introduction

A program slice [32, 22] is constructed by deleting from a program those statements and predicates which cannot affect the value of a set of variables, V, at some point within the program, i. The pair (V, i) is known as the slicing criterion.

Several authors have shown how slicing may be used to aid the task of program comprehension [13, 19, 20, 23]. Slicing assists program comprehension because of the way it abstracts from a program a thread (the slice) which captures the effect of the original upon some chosen sub domain of its overall computation.

For example, consider the program and its slice depicted in Figure 1. The original program, p, is sliced with respect to the final value of the variable `s`, producing a slice, p', which contains those parts of the original program which may affect the final value of the variable `s`. In order to understand the effect of the original program upon `s`, there is clearly no need to consider parts of the program which could not affect `s`. Slicing thus ensures that comprehension effort is not wasted.

The restriction to command deletion is required when slicing is applied to the problem of debugging [26], because a slice is deemed to contain the statement(s) which cause some error to occur and which the debugging process attempts to locate and correct. However, it will be shown that a less restrictive form of slicing, called 'amorphous slicing', is more appropriate when the application is program comprehension, rather than debugging.

In order to define amorphous slicing as a variant of traditional slicing, a projection framework is introduced for defining and comparing approaches to program specialisation such as slicing. Traditional static and conditioned slicing are reformulated within this framework and amorphous versions are introduced in

`scanf("%d",&n);`	`scanf("%d",&n);`
`s=0;`	`s=0;`
`p=1;`	
`while (n>0)`	`while (n>0)`
`  {`	`  {`
`  s=s+n;`	`  s=s+n;`
`  p=p*n;`	
`  n=n-1;`	`  n=n-1;`
`  }`	`  }`
p: Original Program	p': Slice for `s`

Figure 1: A Program and one of its slices

a natural way. The framework facilitates comparison between the various paradigms.

The contribution of the paper can be summarised as follows:

- A theoretical framework of program projection is introduced, facilitating comparison of slicing paradigms.
- A new form of slicing, amorphous slicing, is proposed. Amorphous slicing is more suited to program comprehension problems, while traditional slicing is more suited to debugging.
- The projection framework is used to define amorphous static and amorphous conditioned slicing and to compare these two new forms of slicing with their traditional counterparts.

The rest of this paper is organised as follows: Section 2 introduces the projection framework. Sections 3 and 4 show how Weiser's static slice [30, 32] and the conditioned slice of Canfora, Cimitile, DeLucia and Munro [7, 10, 9, 14, 13] can be formulated as program projections. Section 5 argues that for program comprehension applications, slicing's syntactic restriction to command deletion, should be relaxed. This gives rise to a form of slicing — amorphous slicing— which shares slicing's semantic restriction, but drops its syntactic restriction. Sections 6 and 7 introduce definitions of amorphous static and conditioned slicing, in terms of the projection framework introduced in Section 2, and use the framework to establish a few fundamental results concerning amorphous slicing.

2 Program projection

A program projection is constructed with respect to a measure of simplicity, S and an equivalence relation on programs $\sim$.

2.1 Simplicity

Program projection is concerned with program simplification. The process of simplification will need to be automated (or at least, semi–automated), if it is to be useful and therefore the simplicity measure should be a computable (partial) ordering on programs. For different applications of projection, different simplicity measures will be appropriate. For example, the 'syntactic subset' simplicity measure, implicit in traditional slicing is appropriate for debugging applications. As will be seen, however, a new, relaxed, simplicity measure is more appropriate for program comprehension.

Definition 1 (Simplicity Measure) A *simplicity measure* is a pair $(F, \sqsubseteq)$, where F is a total function from programs to a set which is partially ordered by $\sqsubseteq$.

2.2 Equivalence

The term 'program projection' has been chosen because the process of projection preserves a projection of the original program's semantics. A program and its projection are therefore related to one another by an equivalence relation, $\sim$, on programs.

2.3 Projection

Given a simplicity measure S and an equivalence relation $\sim$, a projection of a program p with respect to $(\sim, S)$ is any program which is as simple as p with respect to S and which is equivalent to p with respect to $\sim$.

Definition 2 ($(\sim, S)$ Projection) Given a simplicity measure $S = (F, \sqsubseteq)$, and an equivalence relation $\sim$, a program p is a $(\sim, S)$ *projection* of a program q if and only if $F(p) \sqsubseteq F(q)$ and $p \sim q$.

Let $[p]_\sim$ denote the set of all programs in the same equivalence class as p according to the equivalence relation $\sim$. Given an equivalence relation $\sim$, many transformations map a program p to one of the programs in $[p]_\sim$. Such transformations shall be referred to as '$\sim$ preserving'. In each equivalence class produced by $\sim$, there will be a set of programs which are simplest. The term $(\sim, S)$ *minimal* program shall be used to describe any program which is in an equivalence class containing no simpler program.

Definition 3 ($(\sim, S)$ Minimality) Given a simplicity measure $S = (F, \sqsubseteq)$, and an equivalence relation $\sim$, the program p is $(\sim, S)$ *minimal* if and only if $\neg\exists q \in [p]_\sim . F(q) \sqsubset F(p)$.

A $(\sim, S)$ *minimal projection* of p is any $(\sim, S)$ minimal program in $[p]_\sim$.

Definition 4 ($\sim S$ Minimal Projection) Given a simplicity measure $S = (F, \sqsubseteq)$, and an equivalence relation $\sim$, a program p is a $(\sim, S)$ *minimal projection* of a program q if and only if p is $(\sim, S)$ minimal program and p is a $(\sim, S)$ projection of q.

3 Static slicing

Traditional approaches to slicing involve a single simplifying transformation, namely, the deletion of commands from a program, or, more precisely, the deletion of nodes from a program's Control Flow Graph (CFG) representation[1]. It is helpful in discussing slicing to uniquely number nodes of the CFG. These node numbers will often be referred to as 'line numbers' because, in representing the text of a program, each CFG node will occupy a single line.

The simplicity measures involved in all existing approaches to slicing are therefore identical. Existing slicing definitions are distinguished by the equivalence on programs they preserve.

3.1 Syntactic subset simplicity

The simplicity measure used in traditional program slicing shall be termed 'syntactic subset simplicity' in this paper to distinguish it from the amorphous simplicity measure introduced later on.

Definition 5 (Syntactic Subset Simplicity) Let F be mapping from programs to a set of line–number $\times$ command pairs, where $(l, c) \in F(p)$ if and only if the program p contains the command c at line number[2] l, and where for all p, no line number appears more than once in $F(p)$. The *syntactic subset simplicity* measure, $\sqsubseteq_{TS}$, is $(F, \subseteq)$.

Weiser's original definition of a slice [30] has come to be known as the static backward slice after the introduction of the dynamic [25] and forward [21] slicing paradigms. Weiser's definition can be reformulated within the projection framework using the syntactic subset simplicity measure, $\sqsubseteq_{TS}$. The only remaining question is therefore: "what is the equivalence, $\sim$, preserved by static slicing?".

[1] The original data flow equation algorithm introduced by Weiser [30], the Program Dependence Graph algorithm introduced by Ottenstein and Ottenstein [27] and Horwitz et al [22] and the the parallel algorithm introduced by Danicic et al [12] all have the Control Flow Graph as their starting point. In the case of Weiser and Danicic et al, the slice is constructed by deleting nodes of the CFG. In the case of Ottenstein and Ottenstein and Horwitz et al, it is constructed by deleting nodes of the PDG. However, the PDG is constructed from the CFG and, but for a few minor exceptions, the PDG and CFG share the same node set.

[2] Every node of the Program Dependence Graph shall be assigned [27] a unique line number as is conventional when describing slices of programs [17, 21].

3.2 The static equivalence relation

The equivalence preserved by static backward slicing is based upon the concept of a state trajectory; a sequence of line–number $\times$ state pairs, mapping m to (l, s) if, after m command executions the state is s, and the next command to be executed is at line number l. Weiser [32] defines 'state trajectory projection' as follows:
Let $T = (t_1, t_2, \ldots, t_k)$ be a state trajectory, n any node and s any function from names to values.

Definition 6 ($Proj'$)

$$Proj'_{(V,i)}((n, s)) = \begin{cases} \lambda & \text{if } n \neq i \\ (n, s \mid V) & \text{if } n = i. \end{cases}$$

Where $s \mid V$ is s restricted to domain V, and λ is the empty string.

$Proj'$ is now extended to entire trajectories:

Definition 7 ($Proj$)

$$Proj_{(V,i)}(T) = Proj'_{(V,i)}(t_1) \ldots Proj'_{(V,i)}(t_k)$$

Using Weiser's definition of projection, traditional (static) equivalence is defined as follows:

Definition 8 (Static Equivalence) Given two programs p and q, a line number i, and a set of variables K, p is *traditionally statically equivalent* to q, written $p\ \mathcal{T}^{(K,i)}\ q$ if and only if for all states σ, when the execution of p in σ gives rise to a trajectory T and the execution of q in σ gives rise to a trajectory T', then $Proj_{(K,i)}(T) = Proj_{(K,i)}(T')$.

Observe that, as static equivalence is parameterized by K and i, definition 8 describes a *class* of equivalence relations based upon the choice of K and i. This reflects that fact that each slicing criterion yields a slice which preserves a *different* projection of the semantics of the program from which it is constructed.

It is easy to verify that $\mathcal{T}^{(K,i)}$ is Reflexive, Transitive and Symmetric, and is therefore an equivalence relation on programs. Instantiating definition 8 of traditional static equivalence for the equivalence relation $\sim$, and definition 5 of syntactic subset simplicity for S in definition 2 of $(\sim, S)$ projection yields the definition of a $(\mathcal{T}^{(K,i)}, \sqsubseteq_{TS})$ projection.

Observe that p is a $(\mathcal{T}^{(K,i)}, \sqsubseteq_{TS})$ projection of q if and only if p is a static slice of q with respect to the slicing criterion (K, i). This is because $p\ \mathcal{T}^{(K,i)}\ q$ means that the values of variables in K immediately prior to the execution of the command at i are identical in both p and q, and $\sqsubseteq_{TS}$ captures the 'command deletion' method of simplification employed by traditional approaches to program slicing. This observation allows traditional static slicing to be reformulated as an instance of program projection, yielding definition 9 below:

Definition 9 (Traditional Static Slice) q is a *traditional static slice* of p with respect to the slicing criterion (K, i) iff q is a $(\mathcal{T}^{(K,i)}, \sqsubseteq_{TS})$ projection of p.

For example, consider the program and its slice depicted in Figure 1. The slice and the original program are $\mathcal{T}^{(\{s\},10)}$ equivalent, and the slice, p', is simpler than the original, p, according to $\sqsubseteq_{TS}$. That is, $p' \sqsubseteq_{TS} p$.

4 Conditioned slicing

Conditioned slicing was introduced by Canfora, Cimitile, DeLucia and Munro [7, 11, 13]. A similar approach (involving conditions) was introduced independently by Field, Tip and Ramalingham [16] and by the present authors [19]. The conditioned slicing criterion is an extension of the static slicing criterion introduced by Weiser [32].

The conditioned slicing criterion includes the set of states in which the program is to be executed, allowing the programmer to specify, not only the variables of interest, but also the initial conditions of interest. This also has the advantage that it allows for additional simplification during slice construction. The ability to analyse a program by cases (one per condition) using conditioned slicing makes it attractive as a supporting technique for program comprehension.

In order to capture conditioned slicing within the projection framework, it will be necessary to specify the equivalence relation preserved by the process of conditioned slicing.

Definition 10 (Conditioned Equivalence) Let K be a set of variables, i a line number in two programs, p and q and let Σ be a set of states. Then p is *conditioned equivalent* to q, written $p\ \mathcal{C}_{\Sigma}^{(K,i)}\ q$, if and only if for all states, σ in Σ, $Proj_{(K,i)}(T) = Proj_{(K,i)}(T')$, where the execution of p in σ gives rise to a state trajectory T and the execution of q in σ gives rise to a state trajectory T'.

The definition of conditioned slicing is obtained by combining the definition of conditioned equivalence with the syntactic subset simplicity given in definition 5. This reformulation[3] of conditioned slicing, within the projection framework is given in definition 11 below.

Definition 11 (Traditional Conditioned Slice) Let K be a set of variables, i a line number in two programs, p and q and let Σ be a set of states. Then q is a *traditional conditioned slice* of p iff q is a $(\mathcal{C}_{\Sigma}^{(K,i)}, \sqsubseteq_{TS})$ projection of p.

As DeLucia et al [13] point out, conditioned slicing subsumes static slicing. In the projection framework this is easy to verify, as by definition of conditioned and static equivalence

$$\forall K, i, S.\mathcal{T}^{(K,i)} \subseteq \mathcal{C}_{S}^{(K,i)}.$$

That is, the conditioned equivalence relation is a weaker equivalence relation than the static equivalence relation. Also, in the limiting case where S is the set of all states

$$\forall K, i.\mathcal{T}^{(K,i)} = \mathcal{C}_{S}^{(K,i)}.$$

A similar argument demonstrates that conditioned slicing subsumes Venkatesh's [29] quasi–static slice.

Traditional static slicing often creates little simplification in practice. For example, consider the program in Figure 2. If a static slice is constructed for the variable **z** at the end of the program, then the smallest possible static slice an algorithm may construct, according to definition 9 of a static slice, will be the whole program.

The observation that static slices tend to be rather large lead to the introduction of dynamic slicing by Korel and Laski [25]. For a dynamic slice, the static slicing criterion is extended to include the entire input sequence supplied to the program. For the program in Figure 2 this will allow one or other of the two assignments to **z** to be removed in constructing a dynamic slice, because only one or other will be executed.

However, although the dynamic slicing criterion does allow for the construction of thinner slices than the static criterion, it does so at the expense of making slicing criteria highly specific. This is not a problem when slicing is applied to the problem of debugging, because the entire input sequence which causes an error to occur will be known. It can be rather restrictive, however, when slicing is applied to the problem of program comprehension.

[3] In De Lucia's formulation a conditioned slicing criterion is a 4–tuple, (Υ, π, i, K), where π is a predicate logic formula, the free variables of which are Υ. K and i play the same role as they do in definition 10 presented here. Υ and π define the set of states $\{\sigma \mid \pi(\sigma)\}$, thus the conditioned criterion can be captured by a set of states (Σ in definition 10 denoting Υ and π in De Lucia's formulation), a set of variables (K) and a point in the original program (i).

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

Figure 2: A Simple Candidate for Conditioned Slicing

Conditioned slicing provides a convenient compromise between the poles of static and dynamic slicing. For example, there are a *set* of possible inputs to the program in Figure 2 which cause statement 4 to be ignored. While dynamic slicing may only capture *one* of these inputs for each slice constructed, conditioned slicing can capture them *all* with a single condition, namely $x \leq y$.

5 Program comprehension and command deletion

The traditional conditioned slicing criterion is a modification of the traditional static slicing criterion. This modification alters the equivalence relation preserved during the slicing process, but not the simplicity measure; like static slices, traditional conditioned slices are constructed using syntactic subset simplicity, restricting any implementation to the sole transformation of command deletion.

This restriction is required when slicing is applied to debugging [26, 33, 31], because the slice helps to locate the line(s) containing the defect which causes a bug to occur. When slicing is applied to program comprehension however, the restriction to command deletion becomes a nuisance, as it may weaken the simplification power of the slicer.

This section introduces amorphous simplicity; an alternative to the syntactic subset simplicity measure. Using the amorphous simplicity measure, amorphous versions of static and conditioned slicing can be defined. These definitions free the traditional approaches from the restriction to command deletion, thereby increasing their applicability to program comprehension.

A natural choice for a simplicity measure is that p is simpler than q if it contains fewer statements. Such a definition of simplicity clearly depends upon how commands are to be counted. A sensible approach, in keeping with work on slicing, is to count the nodes in the CFG [21, 27].

The amorphous simplicity measure, $\sqsubseteq_{AS}$, is captured in definition 12 below:

Definition 12 (Amorphous Simplicity) Let $F(p)$ be the number of nodes in the PDG of p. The amorphous simplicity, $\sqsubseteq_{AS}$, is the pair $(F, \leq)$.

Once the restriction to command deletion is lifted, there are a plethora of replacement simplicity measures from which to choose. The amorphous simplicity measure described above is only one possible alternative to command deletion. It could be replaced by a measure which counts expressions, one which counts side effects or one which counts the number of occurrences of some set of chosen language primitives. In each case, the resulting amorphous slicing process will tend to reduce the number of these syntactic features present in the program, whilst preserving its effect upon the slicing criterion.

Falcon and Blazy [6] apply partial evaluation [5] to the program comprehension problem. In the work of Falcon and Blazy, the approach is applied to procedural programs, with the aim is of making the program more understandable. The work reported here raises similar questions to theirs, namely:

> "Which transformations simplify human understandability?"

rather than

> "Which transformations improve computer performance?"

One attractive feature of the projection framework is the way the simplicity measure can be viewed as any well-defined code–level metric [15, 28].

In the following two sections amorphous simplicity will be used to construct amorphous definitions of static and conditioned slicing.

6 Amorphous static slice

An amorphous slice shares with its traditional counterpart the property that the slicing process preserves a projection of the semantics of the original program. However, amorphous slicing differs from traditional slicing in the admissible transformations which may be brought to bear in slice construction.

Amorphous static slicing can be formally defined using the projection framework introduced in section 2. This yields definition 13 below:

Definition 13 (Amorphous Static Slicing) q is an *amorphous static slice* of p with respect to the slicing criterion (K, i) if and only if q is a $(\mathcal{T}^{(K,i)}, \sqsubseteq_{AS})$ projection of p.

Consider the example program fragments in Figure 3. All the fragments are $\mathcal{T}^{(\{x\},5)}$ equivalent to

`1 {`	`1 {`	`1 {`
	`2 y=10;`	`2 z=-1;`
`3 x=2;`	`3 x=y-8;`	`3 x=z+3;`
		`4 y=10;`
`5 }`	`5 }`	`5 }`
Program 1	Program 2	Program 3

Figure 3: Examples of Amorphous Static Slices

one another, but none is a traditional static slice of any of the others, as none is a syntactic subset of any of the others. However, since Program 1 contains fewer commands than Program 2, which contains fewer commands than Program 3, it can be concluded that Program 1 is a $(\mathcal{T}^{(\{x\},5)}, \sqsubseteq_{AS})$ projection of Program 2 and that Program 2 is a $(\mathcal{T}^{(\{x\},5)}, \sqsubseteq_{AS})$ projection of Program 3. Furthermore, Program 1 is a $(\mathcal{T}^{(\{x\},5)}, \sqsubseteq_{AS})$ minimal projection of Programs 2 and 3 (and of itself), because no other $\mathcal{T}^{(\{x\},5)}$–equivalent program has fewer commands.

Amorphous static slicing is simply traditional static slicing with a relaxed simplicity measure, allowing, for example, Program 1 of Figure 3 to be regarded as a slice of Programs 2 and 3 of Figure 3.

6.1 Amorphous static slicing subsumes traditional static slicing

Traditional static slicing with respect to a criterion (K, i) can only delete commands from p. By definition 9 of traditional static slicing, this process is $\mathcal{T}^{(K,i)}$–equivalence preserving. As the only program transformation permitted is command deletion, there will be at most the same number of commands in the traditional static slice as the original, therefore any implementation of traditional static slicing is also an implementation of amorphous static slicing. More formally, amorphous static slicing subsumes traditional static slicing because $\sqsubseteq_{TS} \subseteq \sqsubseteq_{AS}$.

Consequently there will always be an amorphous static slice which is at least as thin as the static slice constructed for the same slicing criterion (equivalence relation). Of course, there will usually be an amorphous static slice which is *thinner* than the corresponding static slice, and this makes amorphous slicing attractive as a tool for program comprehension.

6.2 Minimal amorphous static slicing is non–computable

The close correspondence between amorphous and traditional static slicing suggests that minimal amorphous static slicing will not be generally computable. This is to be expected as only the *syntactic* component of the definition of a traditional static slice has been altered, leaving the *semantic* component intact.

Proposition 1 $(\mathcal{T}^{(K,i)}, \sqsubseteq_{AS})$ minimal projections (amorphous static slices) are not generally computable.

Proof
Consider the program fragment, P, below, where `f` and `g` are arbitrary functions which have no effect upon the global variable `x`:

```
1    {
2    scanf("%d",x);
3    if (f(x) != g(x))
4       z=1;
5    }
```

Now, consider the $(\mathcal{T}^{(\{z\},5)}, \sqsubseteq_{AS})$ minimal projection, S, constructed from P. This minimal projection can be defined as follows:

$$S = \begin{cases} P & \text{if } \neg(\texttt{f} \equiv \texttt{g}) \\ \{\} & \text{if } \texttt{f} \equiv \texttt{g} \end{cases}$$

Thus, if there existed an algorithm for constructing minimal projections, then this algorithm could be used to decide whether or not two arbitrary functions are equivalent. As this problem is known to be undecidable, it must be concluded that constructing $(\mathcal{T}^{(K,i)}, \sqsubseteq_{AS})$ minimal projections is also not generally computable.

The non–computability of minimal amorphous slices means that, as with traditional slicing algorithms, amorphous slicing algorithms can only be expected to construct good approximations. The formal definition of the simplicity measure will be useful in examining just how well an algorithm achieves this, as it orders programs according to their simplicity, and could therefore be extended naturally to an ordering on slicing algorithms.

7 Amorphous conditioned slicing

Conditioned slicing is ideal for program comprehension situations, because it supports case analysis. The cases are supplied in the form of conditions. Conditioned slices are also at least as thin as corresponding static slices, because they are tailored to a specific condition.

The 'thinness' of conditioned slicing may be further increased if the restriction to command deletion is removed. This creates a definition of 'amorphous conditioned slicing'.

Definition 14 (Amorphous Conditioned Slice)
Let K be a set of variables, i a point in a program p

and Σ be a set of states. q is an *amorphous conditioned slice* of p with respect to the slicing criterion (K, i, Σ) if and only if q is a $(\mathcal{C}_{\Sigma}^{(K,i)}, \sqsubseteq_{AS})$ projection of p.

Amorphous conditioned slicing is related to traditional conditioned slicing in precisely the same way that amorphous static slicing is related to the traditional static slicing. That is, amorphous conditioned slicing subsumes traditional conditioned slicing, because $\sqsubseteq_{TS} \subseteq \sqsubseteq_{AS}$. In practice this means that an algorithm which produces traditional conditioned slices also produces amorphous conditioned slices, although the converse in not true. As amorphous conditioned slicing subsumes traditional conditioned slicing and traditional conditioned slicing subsumes traditional static slicing, it must be concluded that amorphous conditioned slicing is not generally computable.

As with amorphous static slicing, amorphous conditioned slicing allows for greater simplification than its traditional counterpart, further improving the applicability of the conditioned slicing paradigm to the problem of program comprehension.

For example, consider the `atoi` program fragment in Figure 4, taken from page 61 of [24]. The program fragment converts the ASCII string `s` into an integer, storing the result in `r`. A natural way to analyse this program fragment, would consist of considering its behaviour under three initial conditions: where the input string, `s`, is empty, where it starts with a 'minus' sign and where it starts with a 'plus' sign. Using amorphous conditioned slicing, specialised versions of the `atoi` program can be constructed for each case. These are depicted in Figure 4.

Using traditional static slicing, no simplification could be produced (every statement contributes to the final value of `r`). Amorphous static slicing improves upon this situation a little. For example, the effect of the assignment to `sign` at line 2 of the original program can be percolated through the program to its use at line 6. Traditional conditioned slicing also improves upon traditional static slicing, because, exploiting knowledge of the initial state, it will be able to remove the loop at line 1 of the original program. However, traditional conditioned slicing is hampered by the restriction to command deletion, for example, preventing it from exploiting knowledge of the initial state to simplify the expression at line 2 of the original program. Amorphous conditioned slicing brings the full power of the conditioned approach to bare upon the problem, released from the restriction to command deletion.

7.1 Implementation

This paper introduces amorphous slicing without discussing how amorphous slices might be computed. Some initial progress towards this goal was presented (semi–formally) in [18], but the algorithm presented here produces only amorphous static slices (not amorphous conditioned slices), and the algorithm applies only to a simple language, with a few admissible transformations, and with respect to a specific form of analysis (analysing a program's robustness).

Early experimental work indicates that it is likely that amorphous conditioned slices can be computed using a mixture of Partial Evaluation (in the Falcon and Blazy style [6], procedural transformation [4] and traditional slicing. While amorphous static slicing could be achieved with slicing and transformation alone, it seems pointless to do so, as amorphous conditioned slicing subsumes amorphous static slicing.

Fortunately, some slicing problems turn out to be easier to address in the amorphous paradigm. For example, consider the traditional problem of slicing programs in the presence of `goto` statements, which has been addressed by several authors [1, 3, 8]. As a `goto` statement is neither a predicate nor an assignment, it fails to be included in any slice using conventional algorithms for traditional static program slicing [30, 27, 22, 12]. The published resolutions of this problem produce slices which are either not syntactic subsets of the original [8] or produce unnecessarily large slices [8, 3, 1].

In the amorphous paradigm the problem of slicing in the presence of `goto` statements disappears; using Ascroft and Manna's algorithm [2], `goto` statements can be reformulated as loops, allowing traditional static slicing algorithms to be applied (albeit to produce amorphous static slices).

8 But *is* this slicing?

It is natural to ask

> "Is amorphous slicing *really* slicing?"

Weiser [30] pointed out that, for some languages, it will not always be possible for a slice to be a pure syntactic subset of the original program. For example, where a `then` branch of a conditional is removed, whilst the `else` part remains, the slice may not be valid in the syntax of some programming languages.

However, notwithstanding these minor linguistic irritations, slicing algorithms have, hitherto, sought to preserve a syntactic subset of the original program by removing only syntactic sub components during the slicing process.

<table>
<tr><td><pre>
1   for(i=0;isspace(s[i]);i++);
2   sign=(s[i]=='-')?-1:1;
3   if(s[i]=='+' || s[i]=='-') i++;
4   for(n=0;isdigit(s[i]);i++)
5   n = 10*n + (s[i]-'0');
6   r=sign*n;
7
</pre></td><td><pre>

   for(i=1,n=0;isdigit(s[i]);i++)
     n=10*n+(s[i]-'0');
   r=-n;
7
</pre></td></tr>
<tr><td>The atoi Program</td><td>Slice for $(\{\sigma \mid \sigma(s[0]) = '-'\}, \{r\}, 7)$</td></tr>
<tr><td><pre>

   r=0;
7
</pre></td><td><pre>

   for(i=1,n=0;isdigit(s[i]);i++)
     n=10*n+(s[i]-'0');
   r=n;
7
</pre></td></tr>
<tr><td>Slice for $(\{\sigma \mid \sigma(s) = ""\}, \{r\}, 7)$</td><td>Slice for $(\{\sigma \mid \sigma(s[0]) = '+'\}, \{r\}, 7)$</td></tr>
</table>

Figure 4: Analysis by Cases Using Amorphous Conditioned Slicing

If the reader takes the view that 'slice' refers to the manner in which syntactic sub components are 'sliced away' from a program, then it will not be possible to maintain the view that amorphous slicing is really slicing in the strict sense of the word. However, the authors believe that the crucial insight behind all approaches to slicing is the way in which they preserve a *projection* of the original program's semantics. The important property of slicing, relied upon by all its applications, is the way it slices away *semantic* sub components of the original program. Amorphous slicing is thus slicing in the semantic sense, but is not slicing in the syntactic sense.

This is a rather philosophical argument. It remains to be seen whether amorphous slicing will be accepted as a variation of traditional slicing, or whether both amorphous slicing and traditional slicing will be regarded as orthogonal instances of a more general concept of program projection.

9 Conclusion

This paper has introduced a form of syntactically unrestricted slice called an amorphous slice. An amorphous slice, like a traditional slice, preserves a projection of the semantics of the original program from which it is constructed. Unlike traditional slicing, however an amorphous slice is constructed using any simplifying transformation which preserves this projection.

Two forms of amorphous slice are introduced; the static and the conditioned amorphous slice. These are simple variations upon the traditional slicing theme, but the amorphous nature of the new forms of slice makes them thinner and therefore more suited to program comprehension than their traditional counterparts.

In order to constrain the potential explosion of program slicing paradigms, a theoretical framework, the program projection framework, is introduced. It is shown that all forms of slicing discussed in the paper fit into this framework and that it facilitates comparison between these paradigms.

10 Future work

This paper has introduced a new form of slicing, amorphous slicing, but has remained largely silent on the subject of its implementation in practical slicing tools. The amorphous conditioned slices contained in Figure 4 were constructed (by hand) using state propagation, traditional slicing and a simple set of transformations [18]. More work is required to unify work on program transformation (such as the maintainers' assistant project [4]) with work on traditional program slicing, to provide a more general algorithm for constructing amorphous conditioned slices.

The authors believe that the projection framework introduced in this paper will turn out to have wider theoretical applications than merely casting amorphous slicing within the realm of slicing in general. The authors have, for example, managed to fit dynamic and quasi–static slicing into the framework and this process reveals some interesting insights into the definition of a dynamic slice and its relationship to static and conditioned slicing.

Work is in progress to capture all currently existing definitions of program slicing within the projection framework. The goal of this work is to provide a seamless theory which allows results concerning program simplification paradigms to be stated and proved simply and elegantly and which facilitates a comparison of paradigms and algorithms for program specialisation.

Acknowledgements

The authors would like to thank Dan Simpson, Ross Paterson and Siva and Yoga Sivagurunathan for their comments on earlier versions of this paper and Andrea De Lucia for many useful discussions on conditioned slicing. The authors would also like to thank the anonymous referees for their most helpful suggestions which also improved the content and presentation of the paper.

References

[1] Agrawal, H. 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, 29(6), June 1994.

[2] Ashcroft, E., and Manna, Z. The translation of `goto` programs into `while` programs. In *IFIP Congress* (Amsterdam, 1971).

[3] Ball, T., and Horwitz, S. Slicing programs with arbitrary control–flow. In 1^{st} *Conference on Automated Algorithmic Debugging* (1992), P. Fritzson, Ed., Springer, pp. 206–222. Also available as Uniersity of Wisconsin–Madison, technical report (in extended form), TR-1128, December, 1992.

[4] Bennett, K., and Ward, M. Formal methods to aid the evolution of software. *Journal of Software Engineering and Knowledge Engineering 5*, 1 (1995), 25–47.

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

[6] Blazy, S., and Falcon, P. Formal specification and prototyping of a program specializer. In *TAPSOFT'95, Springer LNCS 915* (Aarhus, 1995), pp. 666–680.

[7] Canfora, G., Cimitile, A., De Lucia, A., and Lucca, G. A. D. Software salvaging based on conditions. In *International Conference on Software Maintenance (ICSM'96)* (Victoria, Canada, Sept. 1994), IEEE, pp. 424–433.

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

[9] Cimitile, A., De Lucia, A., and Munro, M. Identifying reusable functions using specification driven program slicing: a case study. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'95)* (Nice, France, 1995), pp. 124–133.

[10] Cimitile, A., De Lucia, A., and Munro, M. Qualifying reusable functions using symbolic execution. In *Proceedings of the* 2^{nd} *working conference on reverse engineering* (Toronto, Canada, 1995), IEEE Press, pp. 178–187.

[11] Cimitile, A., De Lucia, A., and Munro, M. A specification driven slicing process for identifying reusable functions. *Software maintenance: Research and Practice 8* (1996), 145–178.

[12] Danicic, S., Harman, M., and Sivagurunathan, Y. A parallel algorithm for static program slicing. *Information Processing Letters 56*, 6 (Dec. 1995), 307–313.

[13] De Lucia, A., Fasolino, A. R., and Munro, M. Understanding function behaviours through program slicing. In 4^{th} *IEEE Workshop on Program Comprehension* (Berlin, Germany, Mar. 1996).

[14] De Lucia, A., and Munro, M. Program comprehension in a reuse reengineering environment. In 1^{st} *Durham workshop on program comprehension* (Durham University, UK, July 1995), M. Munro, Ed.

[15] Fenton, N. E. *Software Metrics: A Rigorous Approach.* Chapman and Hall, 1990.

[16] Field, J., Ramalingam, G., and Tip, F. Parametric program slicing. In 22^{nd} *ACM Symposium on Principles of Programming Languages* (San Francisco, CA, 1995), pp. 379–392.

[17] Gallagher, K. B., and Lyle, J. R. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering 17*, 8 (Aug. 1991), 751–761.

[18] Harman, M., and Danicic, S. Using program slicing to simplify testing. *Journal of Software Testing, Verification and Reliability 5* (Sept. 1995), 143–162.

[19] Harman, M., Danicic, S., and Sivagurunathan, Y. Program comprehension assisted by slicing and transformation. In 1^{st} *UK workshop on program comprehension* (Durham University, UK, July 1995), M. Munro, Ed.

[20] Harman, M., Danicic, S., Sivagurunathan, Y., and Simpson, D. The next 700 slicing criteria. In 2^{nd} *UK workshop on program comprehension* (Durham University, UK, July 1996), M. Munro, Ed.

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

[22] Horwitz, S., Reps, T., and Binkley, D. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems 12* (1990), 26–61.

[23] Jackson, D., and Rollins, E. J. Chopping: A generalisation of slicing. Tech. Rep. CMU-CS-94-169, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, July 1994.

[24] Kernighan, B. W., and Ritchie, D. M. *The C programming language*. Prentice Hall, 1988. Second Edition, (ANSI C).

[25] Korel, B., and Laski, J. Dynamic program slicing. *Information Processing Letters 29*, 3 (Oct. 1988), 155–163.

[26] Lyle, J. R., and Weiser, M. Automatic program bug location by program slicing. In 2^{nd} *International Conference on Computers and Applications* (Peking, 1987), pp. 877–882.

[27] Ottenstein, K. J., and Ottenstein, L. M. The program dependence graph in software development environments. *SIGPLAN Notices 19*, 5 (1984), 177–184.

[28] Shepperd, M. *Foundations of software measurement*. Prentice Hall, 1995.

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

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

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

[32] Weiser, M. Program slicing. *IEEE Transactions on Software Engineering 10*, 4 (1984), 352–357.

[33] Weiser, M., and Lyle, J. R. *Experiments on slicing–based debugging aids*. Empirical studies of programmers, Soloway and Iyengar (eds.). Molex, 1985, ch. 12, pp. 187–197.