Formalizing Executable Dynamic and Forward Slicing

Dave Binkley[1] **Sebastian Danicic**[2] **Tibor Gyimóthy**[3] **Mark Harman**[4] **Ákos Kiss**[3] **Lahcen Ouarbya**[2]

[1]Loyola College
Baltimore MD
21210-2699, USA.

[2]Goldsmiths College
University of London
New Cross, London
SE14 6NW, UK.

[3]Institute of Informatics
University of Szeged
6720 Szeged, Hungary.

[4]Brunel University
Uxbridge, Middlesex
UB8 3PH, UK.

Abstract

This paper uses a projection theory of slicing to formalize the definition of executable dynamic and forward program slicing. Previous definitions, when given, have been operational, and previous descriptions have been algorithmic. The projection framework is used to provide a declarative formulation in terms of the different equivalences preserved by the different forms of slicing.

The analysis of dynamic slicing reveals that the slicing criterion introduced by Korel and Laski contains three inter-woven criteria. It is shown how these three conceptually distinct criteria can be disentangled to reveal two new criteria. The analysis of dynamic slicing also reveals that the subsumes relationship between static and dynamic slicing is more intricate that previous authors have claimed.

Finally, the paper uses the projection theory to investigate theoretical properties of forward slicing. This is achieved by first re-formulating forward slicing to provide an executable forward slice. This definition allows for formal investigation of the relationship between forward and backward slicing.

1. Introduction

This paper is concerned with the formal definitions and properties of dynamic slicing and forward slicing (rather than algorithms for computing them). It employs the projection theory of program slicing introduced by Harman, Danicic, and Binkley [11, 12]. Using this theory, it is possible to study and explain similarities and differences between slicing algorithms. The theory was first used to examine the differences and similarities between amorphous and syntax–preserving forms of slice [11, 12]. This paper uses the projection theory to investigate the nature of dynamic slicing as originally formulated by Korel and Laski [18] and to formalize the definition of executable forward slice.

The results on dynamic slicing reveal that the nature of the dynamic slicing criterion is more subtle than previous authors have claimed [5, 7, 24]. There is no simple two–element subsumation relationship between Korel and Laski dynamic slicing and static slicing. In addition, it is shown that dynamic slicing criteria contain three interwoven concepts; the input, the trajectory, and the iteration count. Previous authors have regarded the input as the only aspect of the dynamic slicing criterion.

The paper also presents results concerning forward slicing. Instead of defining a forward slice as those components of the program that are *affected by* the slicing criterion, a *complement* forward slice is defined as a program whose execution is *unaffected by* the slicing criterion.

Finally, the paper introduces a subsumation relationship that relates the different types of executable backward slicing methods. With the help of the projection framework three orthogonal slicing criteria (input, iteration count, and trajectory equivalence) are identified and their extremes analyzed. The subsumes relationship of the 2^3 so-identified slicing methods is also given.

Following Venkatesh [24], we use the term 'executable' to mean that a slice of program p is itself an executable program whose semantics are a 'subset' of the semantics of p. This is to be contrasted with a non–executable slice, which is merely a set of statements identified as being relevant to the slicing criterion. We prefer executable slicing definitions because they can be investigated formally, in terms of language semantics and also because, for some applications (*e.g.*, testing, restructuring and reuse), slices need to be executable.

The paper makes three novel theoretical contributions:

1. The analysis of Korel and Laski's definition of dynamic slice [18] reveals that three different and orthogonal slicing criteria have been interwoven. By separating these, the paper introduces two, hitherto unstudied, slicing criteria: the iteration count and the trajectory.

2. A unified framework for the formalization of executable backward slicing methods is introduced. The analysis of this framework reveals the precise nature of the subsumes relationship between the variants of dynamic slicing and static slicing.

3. The traditional definition of forward slice [16] defines a forward slice to be merely a set of statements affected by the slicing criterion, rather than an executable subprogram. The paper introduces two definitions of executable forward slice and relates them to both the traditional definition of a forward slice and to the definition of a backward slice.

The rest of the paper is organized as follows: For completeness, Sections 2 and 3 briefly review the projection theory and how it captures syntax–preserving static backward slicing. Section 4 shows how the projection theory can be applied to Korel and Laski's definition of dynamic slicing, with the result that two new criteria are identified. This section uses the analysis to reveal the precise nature of the subsumes relationship between static and dynamic slicing. Section 5 shows how forward slicing can be accommodated by the theory, leading to two new definitions of (executable) forward slice. Finally, Section 6 presents related work, and Section 7, concludes with directions for future work.

2. The Program Projection Theory

The *projection theory* is, in essence, a generalization of program slicing. It is defined with respect to two relations on programs: a *syntactic ordering* and a *semantic equivalence*. The syntactic ordering is simply an ordering relation on programs. It is used to capture the syntactic property that slicing seeks to optimize. Programs that are lower according to the ordering are considered to be 'better'. The semantic relation is an equivalence relation that captures the semantic property that remains invariant during slicing.

Definition 1 (Syntactic Ordering)
A syntactic ordering, denoted by $\lesssim$, is a computable transitive reflexive relation on programs.

Definition 2 (Semantic Equivalence)
A semantic equivalence, denoted by $\approx$, is an equivalence relation on program semantics.

Definition 3 (($\lesssim, \approx$) Projection)
Given syntactic ordering $\lesssim$ and semantic equivalence $\approx$,

Program p is a $(\lesssim, \approx)$ *projection* of program q
iff
$$p \lesssim q \ \wedge \ p \approx q$$

That is, in a projection, the syntax can only improve while the semantics of interest must remain unchanged.

The following definition formalizes the oft-quoted remark: "a slice is a subset of the program from which it is constructed". It defines the syntactic ordering for syntax–preserving slicing. Note that for ease of presentation, it is assumed that each program component occupies a unique line. Thus, a line number can be used to uniquely identify a particular program component. In this paper, only syntax-preserving forms of slicing are considered [11], so all slicing definitions in this paper will share the following Syntactic Ordering.

Definition 4 (Syntactic Ordering)
Let F be a function that takes a program and returns a partial function from line–numbers to statements, such that the function $F(p)$ maps l to c iff program p contains the statement c at line number l. The syntactic ordering, denoted by $\sqsubseteq$, is defined as follows:

$$p \sqsubseteq q \Leftrightarrow F(p) \subseteq F(q).$$

EXAMPLE. In Figure 1, $p' \sqsubseteq p$.

3. Static Slicing

The semantic property that static slicing respects is based upon the concept of a *state trajectory*: The following definitions of *state trajectory*, *state restriction*, $Proj$, and $Proj'$ are extracted from Weiser's definition of a slice [26].

Definition 5 (State Trajectory)
A *state trajectory* is a finite sequence of line–number $\times$ state pairs:

$$(l_1, \sigma_1)(l_2, \sigma_2) \ldots (l_k, \sigma_k)$$

where entry i is (l_i, σ_i) if after i statement executions the state is σ_i, and the next component to be executed is at line number l_i.

Definition 6 (State Restriction)
Given a state, σ and a set of variables V, $\sigma \mid V$ restricts σ so that it is defined only for variables in V:

$$(\sigma \mid V)x = \begin{cases} \sigma\, x & \text{if } x \in V \\ \perp & \text{otherwise} \end{cases}$$

Definition 7 ($Proj'$)
For slicing criterion (V, n), line number l, and state σ,

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

where λ denotes the empty string.

`1 scanf("%d",&n);`	`1 scanf("%d",&n);`
`2 s=0;`	
`3 p=1;`	`3 p=1;`
`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   }`
p: Original Program	p': Slice for $(\{\texttt{p}\}, 9)$

Figure 1. A Program and one of its slices

Definition 8 ($Proj$)
For slicing criterion (V, n) and state trajectory $T = (t_1, t_2, \ldots, t_k)$,

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

For the *slicing criterion* (V, n), which specifies a slice taken with respect to a set of variables V at line number n, $Proj$ extracts from a state trajectory the values of the variables in V at statement n. Using $Proj$, the semantic equivalence for static slicing can now be defined:

Definition 9 (Static Backward Equivalence)
Given two programs p and q, and slicing criterion (V, n), p is *static backward equivalent* to q, written $p\ S^{(V,n)} q$, iff for all states σ, when the execution of p in σ gives rise to a trajectory T_p and the execution of q in σ gives rise to a trajectory T_q, then $Proj_{(V,n)}(T_p) = Proj_{(V,n)}(T_q)$.

EXAMPLE. In Figure 1, $p \quad S^{(\{\texttt{sum}\},9)} \quad p'$ and $p'\ S^{(\{\texttt{sum}\},9)}\ p$.

Because the static slicing semantic equivalence relation is parameterized by V and n, Definition 9 describes a *class* of relations based upon the choice of V and n. This reflects the fact that each slicing criterion yields a slice that respects a different projection of the semantics of the program from which it is constructed. For example, instantiating Definitions 4 and 9 into Definition 3, yields the following:

Definition 10 (Static Backward Slicing)
Program q is a *static backward slice* of program p with respect to the slicing criterion (V, n) iff q is a $(\sqsubseteq, S^{(V,n)})$ projection of p.

EXAMPLE. Program p' shown in Figure 1 is a $(\sqsubseteq, S^{(\{\texttt{sum}\},9)})$ projection of p.

4. Dynamic Slicing

Static slices must preserve a projection of the semantics of the original program for *all* possible program inputs. In certain applications this requirement is too strict. For example, when debugging only a single input is often of interest. Korel and Laski [18] were the first to introduce a dynamic definition of a slice. A dynamic slice need only preserve the effect of the original program upon the slicing criterion for a fixed input. The dynamic paradigm is ideally suited to problems such as bug-location, because a bug is typically detected as the result of the execution of a program with respect to some specific input.

EXAMPLE. Consider once again, the example in Figure 1, but with `p=1` mistakenly coded as `p=0`. Suppose the original program is executed and given the input 1. The value of `p` at the end of the execution is incorrect — it is 0 when it should be 1. The dynamic slice identifies those statements that contribute to the value of the variable `p` when the input 1 is supplied to the program; in this case, just the line `p=0`. Locating the bug (the faulty initialization 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 contrived example as the input causes the `while` loop to go un-executed. However, in general, dynamic slicing improves precision in several ways. Clearly, statements that remain un-executed are not included in a dynamic slice. In addition, statements that are executed and create data and control dependencies may be removed from the slice should these dependencies be subsequently 'overwritten' during the execution. Also dynamic slicing has more precise information concerning the value of array indices and pointer variables, allowing for the more precise determination of data dependencies.

The literature on dynamic slicing includes many different algorithms [1, 2, 10, 17, 18, 19]. Each of these includes a different definition of what makes one program a dynamic slice of another. Furthermore, many of these algorithms do not output executable programs [1, 2, 10, 17]. Rather, they regard a dynamic slice as the collection of statements that have an effect upon the slicing criterion given the chosen input. By contrast, this paper is concerned solely with executable forms of slicing.

Korel and Laski's definition of a dynamic slice is given below. In the definition, n^w denotes the instruction at the w^{th} position in trajectory T (*i.e.*, $T(w)$, which is n). The definition also uses two auxiliary functions on sequences, F and DEL. $F(T, i)$ is the 'front' i elements of sequence T. $DEL(T, \pi)$ is a filtering operation, which takes a predicate π and returns the sequence obtained by deleting elements of sequence T that satisfy π. For a program p, $\bar{p}$ denotes the set of instructions in p. In this definition, and the rest of the paper, we regard the initial state as a convenient and equivalent way of denoting the input to a program.

`1  x=1;`	`1  x=1;`
`2  x=2;`	
`3  if (x>1)`	`3  if (x>1)`
`4    y=1;`	`4    y=1;`
`5  else`	`5  else`
`6    y=1;`	`6    y=1;`
`7  z=y;`	`7  z=y;`
p: Original Program	p': Slice w.r.t. $(\{\mathrm{z}\}, 7)$

Figure 2. A static slice, which is not a KL–slice.

Definition 11 (KL slice [18])
Given a program p, let x be an input to p, T the trajectory obtained from running p on x, w a natural number, and V a set of variables. A *dynamic slice* of p taken with respect to the slicing criterion $C = (x, n^w, V)$ is any executable program p' obtained from p by deleting zero or more statements from p such that when p' is executed on input x it produces a trajectory T' for which there exists an execution position w' and all of the following hold:

(KL1) $F(T', w') = DEL(F(T, w), \lambda t.t \notin \bar{p'})$.

(KL2) For all $v \in V$, the value of v before the execution of instruction $T(w)$ in T equals the value of v before the execution of instruction $T'(w')$ in T'.

(KL3) $T'(w') = T(w) = n$

The problem with Korel and Laski's definition of dynamic slicing is that it is incomparable with the definition of traditional static slicing, *i.e.*, a static slice is not necessarily (an overly large) Korel and Laski style dynamic slice, as one would intuitively expect (conversely, it is trivial that a KL-slice is not necessarily a static slice). Consider the example program p in Figure 2. The program p' is a valid static slice with respect to $(\{\mathrm{z}\}, 7)$ since at line 7 the value of z is 1 for all inputs, just like in p. Unfortunately, p' is not a Korel and Laski style dynamic slice of p with respect to $(\langle\rangle, 7^5, \{\mathrm{z}\})$, because the trajectory of p is (1 2 3 4 7), which is (1 3 **4** 7) after the application of DEL. However, the trajectory of p' is (1 3 **6** 7), which violates (KL1). The restriction KL1 imposes can thus prevent us from choosing a slice from several semantically equivalent subprograms.

Notice that (KL1) is the cause of the incomparability between KL-slicing and static slicing as defined in Definition 10 because (KL1) prevents Korel and Laski style dynamic slicing from begin described using Definitions 7 and 8. To fit KL-dynamic slicing into the framework, we extend Definitions 7 and 8 as follows.

Definition 12 ($\boldsymbol{Proj^{**}}$) For a set of variables V, set of line-number $\times$ natural number pairs P, set of line numbers I, line-number $\times$ natural number pair $n^{(k)}$ and state σ:

$$Proj^{**}_{(V,P,I)}(n^{(k)}, \sigma) = \begin{cases} (n, \sigma|V) & \text{if } n^{(k)} \in P \\ (n, \perp) & \text{if } n^{(k)} \notin P \text{ and } n \in I \\ \lambda & \text{otherwise.} \end{cases}$$

Definition 13 ($\boldsymbol{Proj^{*}}$)
For a set of variables V, set of line-number $\times$ natural number pairs P, set of line numbers I and trajectory T:

$$Proj^{*}_{(V,P,I)}(T) = \prod_{i=1}^{l} Proj^{**}_{(V,P,I)}(n_i^{(k_i)}, \sigma_i),$$

where $\prod_{i=1}^{l} A_i = (A_i)(A_2)\cdots(A_l)$, $T(i) = (n_i, \sigma_i)$, k_i is the number of occurrences of n_i in the first i elements of T, and l is the highest index j in T such that $n_j^{(k_j)} \in P$.

Notice, that $Proj^{**}$ is the counterpart of $Proj'$, but $Proj^{**}$ does not necessarily map $(n^{(k)}, \sigma)$ to λ if $n^{(k)} \notin P$. Using these functions we can give a unified equivalence relation with which we are able to express Korel and Laski's dynamic slicing.

Definition 14 (Unified Equivalence)
For programs p and q, and the sets of line numbers of p and q, denoted $\bar{p}$ and $\bar{q}$ respectively and for a set of states S, set of variables V, set of line-number $\times$ natural number pairs P and set of line-numbers $\times$ set of line-numbers $\rightarrow$ set of line-numbers function X the unified equivalence (U) is defined as follows:

$$p\ U^{(S,V,P,X)}\ q \quad \text{iff} \quad \forall \sigma \in S. Proj^{*}_{(V,P,X(\bar{p},\bar{q}))}(T_p^\sigma) = Proj^{*}_{(V,P,X(\bar{p},\bar{q}))}(T_q^\sigma)$$

where T_r^σ is the trajectory of program r executed on σ.

The roles of the parameters are as follows: S denotes the set of initial states for which the equivalence must hold. This allows us to capture the 'pure' part of the dynamic slicing criterion (the input supplied to the program, or, equivalently the initial state state in which the program is to be executed). The set of variables V is common to all slicing criteria. The set of pairs, P, allows us to capture, and to separate out, the iteration count component of the Korel and Laski dynamic slicing criterion. Finally, the parameter X is a function, used to capture the set of statements which are preserved in trajectory projection by the second rule of Definition 12 of $Proj^{**}$. This rule allows us to keep a 'ghost' of a line-number $\times$ state pair, in the form $(n, \perp)$, for line number n and 'dummy' state $\perp$, in order to fulfill the 'trajectory requirement' of (KL1). We shall use the function

X to either capture the intersection of the two sets of statements from the programs p and q passed to it (where the criterion is to include the (KL1) restriction) or the empty set (where the (KL1) restriction is not included).

By instantiating this definition with appropriate parameters we get a new equivalence relation which captures the semantics of Korel and Laski's dynamic slicing.

Definition 15 (Korel and Laski Style Dynamic Equivalence) For a state σ, set of variables V and a line-number $\times$ natural number pair $n^{(k)}$ the Korel and Laski style dynamic equivalence (D_{KLi}) is defined as follows:

$$D_{KLi}^{(\sigma,V,n^{(k)})} = U^{(\{\sigma\},V,\{n^{(k)}\},\cap)}$$

We shall adopt the notational convention that a 'KL' subscript indicates that a slicing criterion respects Korel and Laski's requirement (KL1) and that an 'i' subscript indicates that a criterion respects Korel and Laski's 'iteration count' requirement. Theorem 1 establishes that Definition 15 faithfully captures Korel and Laski's definition. The rest of the section concentrates on a more detailed examination of the two new criteria: KL1-respecting (which we call trajectory equivalence), and iteration count.

Theorem 1
A program p' is a Korel and Laski style dynamic slice of p with respect to dynamic slicing criterion (σ, n^w, V) iff p' is a $(\sqsubseteq, D_{KLi}^{(\sigma,V,n^{(k)})})$ projection of p, where w is the position of the k^{th} occurrence of n in T_p^σ.

Proof
Here we show the equivalence informally, while the formal proof is given in Appendix A. It is easy to see that the phrase "obtained from p by deleting zero or more statements from it" is equivalent with $\sqsubseteq$. Furthermore, the second and third cases of $Proj^{**}$ correspond to the DEL auxiliary function. Finally, the semantics of (KL1) (and F) are captured by $Proj^*$, the first case of $Proj^{**}$ gives (KL2), and from $Proj^{**}$ and the definition of l in $Proj^*$ follows (KL3). $\square$

From the new definition of Korel and Laski's dynamic slicing, we can identify several orthogonal slicing criteria concepts within its slicing criterion. The traditional view of dynamic slicing is that it is obtained from static slicing by the addition of the input sequence to the static slicing criterion. It turns out that this is not the case for KL dynamic slicing. It is more subtle than that. However, using the projection theory it is possible to tease apart these criterion components.

As noticed earlier, Korel and Laski's dynamic slicing cannot be obtained from the traditional static slicing by the mere addition of the input sequence to the slicing criterion but the (KL1) restriction must also hold. This restriction is captured by the X parameter of the unified equivalence relation and is a hitherto un-revealed criterion.

With the help of the unified equivalence we can now give a new equivalence relation which yields a static slicing comparable with Korel and Laski's dynamic slicing and we can also redefine the traditional static equivalence.

Definition 16 (Korel and Laski Style Static Equivalence and Traditional Static Equivalence)

$$\begin{aligned} S_{KL}^{(V,n)} &= U^{(\mathbf{\Sigma},V,\{n\}\times\mathbf{N},\cap)}, \\ S^{(V,n)} &= U^{(\mathbf{\Sigma},V,\{n\}\times\mathbf{N},E)} \end{aligned}$$

where $\mathbf{\Sigma}$ is the set of all possible states, $\mathbf{N}$ is the set of natural numbers, and for every set of line numbers, x and y, $E(x, y) = \emptyset$.

It is trivial to show that $S^{(V,n)}$ is simply a reformulation of the static backward equivalence given in Definition 9 and it is also clear that $(\sqsubseteq, S_{KL}^{(V,n)})$ is comparable with Korel and Laski's dynamic slicing, *i.e.*, with $(\sqsubseteq, D_{KLi}^{(\sigma,V,n^{(k)})})$, since $\forall \sigma, V, n, k. S_{KL}^{(V,n)} \subseteq D_{KLi}^{(\sigma,V,n^{(k)})}$.

We can also observe that there is another not-yet-discussed concept in the criterion of Korel and Laski's dynamic slicing, which is easier to notice when comparing the unified equivalence-based definitions of the traditional static and KL-slicing. Parameter P gives the point(s) of interest in the trajectory in the form of $n^{(k)}$, which has a slightly different meaning from the n^w component of Korel and Laski's original slicing criterion. While n^w denotes the w^{th} position in the trajectory, which is n, $n^{(k)}$ marks the k^{th} occurrence of n in the trajectory and the exact position is only implicitly given. Theorem 1 shows that these two notations are equivalent, and this allows us to recognize the iteration count as a new type of criterion. Additionally, Definition 16 reveals that by including the concept of iteration count we can express static slicing and Korel and Laski slicing using the same equivalence relation, facilitating further investigation of the subsumation relationship for slicing.

Now that we have identified the orthogonal criterion components (set of initial states, iteration count, (KL1) restriction) we can give the equivalence relations resulting from the various parameterizations of the unified equivalence. For the sake of completeness those relations which have already been presented are repeated in this list:

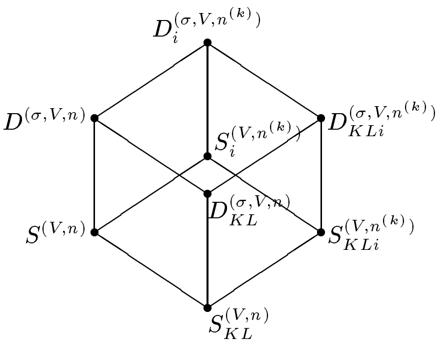

Figure 3. Subsumes relationship between equivalence relations (for any given σ, V, n and k).

$$
\begin{aligned}
S^{(V,n)} &= U^{(\mathbf{\Sigma},V,\{n\}\times\mathbf{N},E)},\\
S_i^{(V,n^{(k)})} &= U^{(\mathbf{\Sigma},V,\{n^{(k)}\},E)},\\
D^{(\sigma,V,n)} &= U^{(\{\sigma\},V,\{n\}\times\mathbf{N},E)},\\
D_i^{(\sigma,V,n^{(k)})} &= U^{(\{\sigma\},V,\{n^{(k)}\},E)}\\
S_{KL}^{(V,n)} &= U^{(\mathbf{\Sigma},V,\{n\}\times\mathbf{N},\cap)},\\
S_{KLi}^{(V,n^{(k)})} &= U^{(\mathbf{\Sigma},V,\{n^{(k)}\},\cap)},\\
D_{KL}^{(\sigma,V,n)} &= U^{(\{\sigma\},V,\{n\}\times\mathbf{N},\cap)},\\
D_{KLi}^{(\sigma,V,n^{(k)})} &= U^{(\{\sigma\},V,\{n^{(k)}\},\cap)}.
\end{aligned}
$$

We remind the reader that D and S denote dynamic and static criteria respectively and that the subscript 'KL' denotes criteria which respect (KL1), while the subscript 'i' denotes criteria which are sensitive to the iteration count.

Figure 3 presents the subsumes relationship between these 8 criteria as a lattice ordered by relational subset. As can be seen, the relationship between static and dynamic slicing is not as straightforward as previous authors have claimed [5, 7, 24].

5. Forward Slicing

Until now, all definitions of slicing considered in this paper have been 'backward' slicing definitions. Horwitz et al. introduced the concept of a *forward* slice [15, 16]:

Definition 17 (Conventional Forward Slice)
A forward slice is constructed from a program, p, with respect to a pair (V, n). It is the set of statements and predicates that are affected by the values of any of the variables in V at n in p.

Unfortunately, this definition of a forward slice leads to slices that are not executable, placing the slices so-defined beyond the scope of the theory introduced here. Fortunately, it is possible to define a form of forward slicing that is faithful to the spirit of forward slicing and for which the slices constructed will be executable programs.

The original definition of forward slicing was based on the dual of the backward slicing algorithm; it simply traverses the edges of the system dependence graph in the opposite direction [16]. No formal semantics for this definition has been given. Rather than the dual of an implementation, the definition provided herein is the dual of the definition. By formalizing the definition of forward slicing, it becomes possible to express the way in which it is a dual of the backward slicing definition.

In the following treatment of forward slicing we shall only consider slices taken at the start or the end of a code fragment, rather than at an arbitrary intermediate point. However, the results can be generalized to cover the case of slicing at intermediate points, though to do so would complicate the exposition.

In the formalization that follows, $\mathcal{M}$ is a semantic meaning function [22]. That is, $\mathcal{M}$ takes a program and returns a state transformation function, which denotes the meaning of the program. Cartwright and Felleisen [6] observed that in program dependence analysis the meaning function is implicitly *lazy*, because dependence analysis treats a program as if it denotes a dataflow computation. Thus, the meaning function, $\mathcal{M}$, of use here is given by lazy semantics. In lazy semantics, some elements of the state may become undefined ($\perp$) while others remain defined. Also, an undefined value at a variable v can be overwritten with a defined value, should v be assigned a proper value after having been involved in a non–terminating computation.

In order to capture forward slicing formally, we require the concept of a 'needed variable' [13, 14]. From the point of view of slicing, the needed variables are those whose initial value is needed in order for a slice to compute correctly.

Definition 18 (Needed Variables)
Variable x *needs* variable y in program p iff there exist two states σ_1 and σ_2 differing only on y were $\mathcal{M}[\![p]\!]\sigma_1 x \neq \mathcal{M}[\![p]\!]\sigma_2 x$

Definition 19 (Needed(x, p))
Needed(x, p) is the set of all variables y such that x needs y in p.

Definition 20 (NeededBy(V, p))
NeededBy(V, p) is the set of all variables x such that for some v in V, x needs v in p.

Consider the simple example fragment in Figure 4. In this example, Needed$(\mathtt{z}, p) = \{\mathtt{y}, \mathtt{a}\}$, Needed$(\mathtt{x}, p) = \{\mathtt{a}\}$ and all other variables (including those not mentioned by the program) need only themselves.

We can now define Static Forward Equivalence in terms of Static Backward Equivalence (Definition 9), and thereby

```
1   x=a;
2   r=r+1;
3   y=y+1;
4   z=x+y;
```

Figure 4. Forward slicing

define Static Forward Slicing in terms of Static Backward Slicing. In so doing we establish a formal link between forward and backward slicing. There is a one-to-one correspondence between the two forms of slicing: for every executable forward slice, q, of a program p taken with respect to criterion C, there is a corresponding backward slicing criterion C' for which q is a backward slice of p with respect to C'.

Definition 21 (Static Forward Equivalence)
Let p be a program and V be a slicing criterion (a set of variables). Program q is static forward equivalent to p with respect to V iff q is static backward equivalent to p with respect to NeededBy(V, p).

Consider the example in Figure 4. Suppose we wish to construct a forward slice with respect to V, where $V = \{a\}$. According to Definition 20, the set NeededBy$(V, p) = \{\mathtt{z}, \mathtt{x}, \mathtt{a}\}$. Therefore, q can only be the set of statements $\{1, 2, 3, 4\}$ or $\{1, 3, 4\}$ since these are the only programs which are static backward equivalent to the original program with respect to NeededBy(V, p). Using the syntax–preserving syntactic ordering (Definition 4) we would therefore choose $\{1, 3, 4\}$ as the executable forward slice with respect to $\{a\}$.

Observe that this form of forward slice is not the traditional forward slice, but it does *contain* the traditional forward slice. This relationship parallels the relationship between backward interprocedural closure slices and backward interprocedural executable slices where the closure slice is contained in the executable slice [3].

Having defined an executable form of forward slice (in terms of a related backward slice), we are now able to define the complement slice. This is also an executable formulation of forward slicing. It will turn out that we shall be able to extract from it, the traditional forward slice. In order to define the complement slice, we need to first define the complement set of variables. This allows us to define complement forward equivalence. By combining complement forward equivalence (as the semantic equivalence) and the traditional syntax–preserving syntactic ordering, we produce a definition of executable forward slicing.

Definition 22 (Complement Variable Set)
Given a program p and a set of variables V, the complement (in p) of V is the set of variables which are mentioned (assigned or referenced) in p but which are not in V.

Definition 23 (Complement Forward Equivalence)
Let p be a program and V be a slicing criterion (a set of variables). Program q is complement forward equivalent to p with respect to V iff q is static backward equivalent to p with respect to any subset of the complement (in p) of NeededBy(V, p).

Consider the example in Figure 4 once more. For this program, the Complement Forward Slice with respect to $\{\mathtt{a}\}$ is $\{2, 3\}$. Observe that this is precisely the complement of the traditional (non executable) forward slice as defined (algorithmically) by Horwitz, Reps and Binkley [16].

Theorem 2 (below) shows that a complement forward slice of a program p with respect to a set of variables V, is a program which behaves identically to p with respect to the variables to which it assigns proper values when both are executed in a state which only differs at the variables in V. That is, altering the initial values of variables in V has no effect upon p and its complement slice (constructed with respect to V). This means that the complement slice captures an executable subprogram which is *unaffected* by the slicing criterion.

A corollary of this result is that the complement of the complement slice is the traditional forward slice. This is because the complement of the complement slice contains statements which (from Theorem 2) *might* potentially be affected by changes to the initial values of the slicing criterion. The complement forward equivalence relation is so–named because it defines an executable forward slice which is the complement of the traditional (non executable) forward slice.

Theorem 2 (Meaning of Complement Slicing) Let p be a program and V a finite set $\{v_0 \cdots v_n\}$ of variables. Let σ and σ' be two states such that σ and σ' differ only upon the elements V and for all $v \in V$, $\sigma' v = \perp$. Then $x \notin$ NeededBy$(V, p) \Rightarrow \mathcal{M}[\![p]\!]\sigma x = \mathcal{M}[\![p]\!]\sigma' x$.

Proof
Since $x \notin$ NeededBy(V, p), $Needed(x, p) \cap V = \emptyset$. Pick any state σ which maps no variables in V to $\perp$. Since $v_0 \notin Needed(x, p)$, there are no two states differing only on v_0 with different final values of x. From this, it follows that

$$\mathcal{M}[\![p]\!]\sigma[v_0 \mapsto \perp]x = \mathcal{M}[\![p]\!]\sigma x.$$

Similarly, since $v_1 \notin Needed(x)$,

$$\begin{array}{rcl} \mathcal{M}[\![p]\!]\sigma[v_0 \mapsto \perp, v_1 \mapsto \perp]x & = & \mathcal{M}[\![p]\!]\sigma[v_0 \mapsto \perp]x \\ & = & \mathcal{M}[\![p]\!]\sigma x. \end{array}$$

Continuing in this way, we see that

$$\mathcal{M}[\![p]\!]\sigma[v_0 \mapsto \perp \cdots v_n \mapsto \perp] = \mathcal{M}[\![p]\!]\sigma x$$

as required. □

Corollary 1 (Traditional Forward Slice)
The complement of the complement forward slice of p with respect to V is the traditional (non executable) forward slice of p with respect to V.

The traditional 'definition' of a forward slice is "the set of statements which may be affected by the slicing criterion". Our formal analysis of forward slicing indicates that there is plenty of scope for ambiguity with the interpretation of 'affected by'. Our meaning function, $\mathcal{M}$ was defined in the traditional denotational style, as a state mapping. This means that to have an *effect*, the value of a criterion variable must affect some element in the *final state*. An alternative, equally plausible definition, is that the value of a criterion variable must affect the value of an *intermediate* computation. There is a similar distinction here to that between weak and strong mutation in mutation testing [27].

Finally, observe that the results presented in this paper bring out the relationship between forward and backward slicing. We are now in a position to express the symmetry between forward and backward slicing as follows:

> "**A complement forward slice is the largest subprogram unaffected by the initial values of variables in the slicing criterion, while the backward slice is the smallest subprogram affected by the final values of variables in the slicing criterion.**"

6. Related Work

Tip [23], and Binkley and Gallagher [4] provide surveys of program slicing. These papers provide a broad picture of slicing technology, tools, applications, definitions, and theory. This paper compliments this work. It shows that backward dynamic slicing and static forward slicing can be placed within the projection framework introduced by Harman et al. [11, 12]. This allows a comparison between the three to be made and extends the set of slicing techniques that can be directly compared. In this respect the aim of the work reported here most closely resembles Venkatesh's earlier work on the semantics of program slicing [24].

Venkatesh provided a formal description of program slicing. His 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 that identify statements and predicates whose computation would become contaminated when some particular variable was initially contaminated.

Venkatesh defined 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. Venkatesh therefore considers 2^3 slicing criteria, some of which had not, at the time, been thought of before (for example the forward dynamic slice).

Canfora et al. [5] showed how conditioned slicing subsumes traditional backward slicing and dynamic slicing. Their argument was based upon a pure notion of dynamic slicing, in which the dynamic slicing criterion is constructed by adding an input sequence to the static criterion. As this paper shows, the relationship between dynamic and static slicing is a little more subtle, when one takes into account Korel and Laski's original formulation of dynamic slicing.

Though many authors have investigated formal properties of static backward slicing [6, 9, 21, 26], there has, hitherto, been no attempt in the literature to give a formal characterization of forward slicing, nor the relationship between Korel and Laski's executable dynamic slicing and other forms of slicing.

7. Conclusion and Future Work

This paper presented results concerning the theory of program slicing. The projection theory of slicing was used to uncover the precise relationship between various forms of dynamic slicing and static slicing and to produce formal and executable definitions of forward slicing. Future work includes encompassing other definitions of slicing, such as decomposition slicing [8], conditioned slicing [5], semantic slicing [25] and barrier slicing [20].

A. Appendix

Here we give the formal proof of Theorem 1.

Proof
We have to show that p' is a $(\sqsubseteq, D_{KLi}^{(\sigma,V,n^{(k)})})$ projection of p, assuming that p' is a Korel and Laski style dynamic slice of p with respect to (σ, n^w, V). The fact that a Korel and Laski style dynamic slice is a syntactic subset of the program from which it is constructed implies $p' \sqsubseteq p$. Reformulating (KL1) gives

$$\prod_{i=1}^{w'} \pi(T_{p'}^{\sigma}(i)) = \prod_{i=1}^{w} \delta(\pi(T_{p}^{\sigma}(i))),$$

where

$$\begin{array}{rcl} \pi((m,s)) & = & (m,\perp), \\ \delta((m,s)) & = & \left\{ \begin{array}{ll} (m,s) & \text{if } m \in p' \\ \lambda & \text{otherwise} \end{array} \right. , \end{array}$$

from which follows

$$\prod_{i=1}^{w'} \delta(\pi(T_{p'}^{\sigma}(i))) = \prod_{i=1}^{w} \delta(\pi(T_{p}^{\sigma}(i))).$$

By substituting $\delta' = \pi \circ \delta$ we get

$$\prod_{i=1}^{w'} \delta'(T_{p'}^{\sigma}(i)) = \prod_{i=1}^{w} \delta'(T_p^{\sigma}(i)).$$

From (KL2) and (KL3) it follows that

$$\delta''(T_{p'}^{\sigma}(w')) = \delta''(T_p^{\sigma}(w)),$$

where

$$\delta''((m, s)) = (m, s|V).$$

Combining the two equations we get

$$\prod_{i=1}^{w'} \Delta_{w'}(i, T_{p'}^{\sigma}(i)) = \prod_{i=1}^{w} \Delta_w(i, T_p^{\sigma}(i)),$$

where

$$\Delta_j(i, (m, s)) = \begin{cases} (m, s|V) & \text{if } i = j \\ (m, \perp) & \text{if } i \neq j \text{ and } m \in p' \\ \lambda & \text{otherwise.} \end{cases}$$

Since, from (KL1) and (KL3) it follows that at position w is the k^{th} occurrence of n in T_p^{σ} and at position w' is also the k^{th} occurrence of n in $T_{p'}^{\sigma}$, it is clear that the above equation is only a reformulation of $p'\ U^{(\sigma, V, \{n^{(k)}\}, \cap)}\ p$.

Conversely, we also have to show that p' is a Korel and Laski style dynamic slice of p with respect to (σ, n^w, V) if p' is a $(\sqsubseteq, D_{KLi}^{(\sigma, V, n^{(k)})})$ projection of p. From $p' \sqsubseteq p$ follows immediately that p' is a syntactic subset of p. By reformulating $p'\ D_{KLi}^{(\sigma, V, n^{(k)})}\ p$ we get

$$\prod_{i=1}^{l'} Proj^{**}_{(V, \{n^{(k)}\}, p')}(n_i'^{(k_i')}, \sigma_i') =$$

$$= \prod_{i=1}^{l} Proj^{**}_{(V, \{n^{(k)}\}, p')}(n_i^{(k_i)}, \sigma_i),$$

where by definition (n_i, σ_i) is $T_p^{\sigma}(i)$, k_i is the number of occurrences of n_i in the first i elements of T_p^{σ} and l is the highest index j such that $n_j^{(k_j)} \in \{n^{(k)}\}$. Respectively, n_i', σ_i', k_i' and l' have similar meanings in $T_{p'}^{\sigma}$. From this follows that l is the position of $n^{(k)}$ in T_p^{σ} and l' is the position of $n^{(k)}$ in $T_{p'}^{\sigma}$. Substituting l with w and l' with w' the above equation implies (KL1), (KL2) and (KL3). $\Box$

References

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

[2] A. Beszédes, T. Gergely, Z. M. Szabó, J. Csirik, and T. Gyimóthy. Dynamic slicing method for maintenance of large C programs. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering (CSMR 2001)*, pages 105–113. IEEE Computer Society, Mar. 2001.

[3] D. W. Binkley. Precise executable interprocedural slices. *ACM Letters on Programming Languages and Systems*, 3(1-4):31–45, 1993.

[4] D. W. Binkley and K. B. Gallagher. Program slicing. In M. Zelkowitz, editor, *Advances in Computing, Volume 43*, pages 1–50. Academic Press, 1996.

[5] G. Canfora, A. Cimitile, and A. De Lucia. Conditioned program slicing. In M. Harman and K. Gallagher, editors, *Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 595–607. Elsevier Science B. V., 1998.

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

[7] C. Fox, S. Danicic, M. Harman, and R. M. Hierons. ConSIT: a fully automated conditioned program slicer. *Software—Practice and Experience*, 34:15–46, 2004. Published online 26th November 2003.

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

[9] R. Giacobazzi and I. Mastroeni. Non–standard semantics for program slicing. *Higher-Order and Symbolic Computation*, 16(4):297–339, 2003. Special issue on Partial Evalution and Semantics-Based Program Manipulation.

[10] R. Gopal. Dynamic program slicing based on dependence graphs. In *IEEE Conference on Software Maintenance*, pages 191–200, 1991.

[11] M. Harman, D. W. Binkley, and S. Danicic. Amorphous program slicing. *Journal of Systems and Software*, 68(1):45–64, Oct. 2003.

[12] M. Harman and S. Danicic. Amorphous program slicing. In 5^{th} *IEEE International Workshop on Program Comprehesion (IWPC'97)*, pages 70–79, Dearborn, Michigan, USA, May 1997. IEEE Computer Society Press, Los Alamitos, California, USA.

[13] M. Harman, C. Fox, R. M. Hierons, L. Hu, S. Danicic, and J. Wegener. Vada: A transformation-based system for variable dependence analysis. In *IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2002)*, pages 55–64, Montreal, Canada, Oct. 2002. IEEE Computer Society Press, Los Alamitos, California, USA. Voted best paper by attendees.

[14] M. Harman, R. M. Hierons, S. Danicic, J. Howroyd, M. Laurence, and C. Fox. Node coarsening calculi for program slicing. In 8^{th} *Working Conference on Reverse Engineering*, pages 25–34, Stuttgart, Oct. 2001. IEEE Computer Society Press, Los Alamitos, California, USA.

[15] S. Horwitz, T. Reps, and D. W. 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.

[16] S. Horwitz, T. Reps, and D. W. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–61, 1990.

[17] M. Kamkar, N. Shahmehri, and P. Fritzson. Interprocedural dynamic slicing. In *Proceedings of the* 4^{th} *Conference on Programming Language Implementation and Logic Programming*, pages 370–384, 1992.

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

[19] B. Korel and J. Rilling. Dynamic program slicing methods. In M. Harman and K. Gallagher, editors, *Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 647–659. Elsevier, 1998.

[20] J. Krinke. Barrier slicing and chopping. In *IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, pages 81–87, Amsterdam, Netherlands, Sept. 2003. IEEE Computer Society Press, Los Alamitos, California, USA.

[21] T. Reps and W. Yang. The semantics of program slicing. Technical Report Technical Report 777, University of Wisconsin, 1988.

[22] J. E. Stoy. *Denotational semantics: The Scott–Strachey approach to programming language theory*. MIT Press, 1985. Third edition.

[23] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, Sept. 1995.

[24] 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.

[25] M. Ward. The formal approach to source code analysis and manipulation. In 1^{st} *IEEE International Workshop on Source Code Analysis and Manipulation*, pages 185–193, Florence, Italy, 2001. IEEE Computer Society Press, Los Alamitos, California, USA.

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

[27] M. R. Woodward and K. Halewood. From weak to strong, dead or alive? an analysis of some mutation testing issues. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, Banff, Canada, July 1988.